EVENTUAL FAULT RECOVERY STRATEGIES FOR BYZANTINE FAILURES

BY

ALEXANDER ZAHDEH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisor:

Professor Nitin Vaidya

Abstract

Byzantine faults in distributed systems can have very destructive consequences for services built on top of these systems but are not commonly tolerated in production systems due to the overhead and scalability limitations with existing approaches such as Byzantine fault tolerance [1]. This work describes a reactive protocol for recovering from Byzantine failures in replicated state machines. In contrast to traditional Byzantine fault tolerance (BFT), which attempts to mask faults, this protocol is designed to allow faults to be exposed to clients but ensures that no client can fork the state of the system by rolling back faulty updates once they are detected. This ensures that, in spite of Byzantine failures, the system will always converge to a consistent state. The system provides a contract to the client called *lapse* consistency that bounds the number of inconsistent reads that can be experienced as a result of the rollbacks that it performs. This system extends prior work on Byzantine detection [2] to provide an integrated system that can not only eventually detect, but also respond to Byzantine faults with provable consistency semantics while preserving many of the important properties of Byzantine detection such as scalability, and responsiveness. We evaluate the overhead of a proof of concept implementation of the system.

To Father and Mother.

Acknowledgments

I would like to thank my parents and my sister who have supported me throughout the process and gave me inspiration and direction. I would also like to thank my adviser Prof. Vaidya for providing valuable discussion and feedback throughout the process. In addition, I would like to thank Lewis Tseng for his valuable feedback and constructive criticism.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

With the scale of modern distributed systems, failures have fast become the common case rather than the exception. Large scale deployments of distributed storage systems must tolerate crash failures as a necessity of the scale at which they operate. More malignant types of failures, such as corrupted, malicious, or colluding nodes are not commonly tolerated in these types of systems such as Cassandra [3], HBase [4], Riak [5], and Voldemort [6] due to the large overhead associated with masking such faults. The most general classification of failures in distributed systems, Byzantine failures, incorporates all of these potential failure scenarios. Byzantine fault tolerance [1] (BFT) is a well known algorithm for masking such failures in a replicated state machine. This algorithm requires that every replica verifies its requests with every other replica in the system through a 3-phase protocol. The cost associated with this protocol has proven too high to deploy in production given that these Byzantine failures are relatively rare in practice.

Though Byzantine faults are rare, the consequences they wreak can be devastating. Byzantine failures in even a single node can often lead to what is known as cascading failures, a situation in which a small subset of the nodes in the system cause faults in a large portion of the system due to errors being propagated. Case studies from major outages such as [7], [8], [9], and [10] have shown that cascading failures have been the cause of major outages in large Internet services. These errors are often caused by the large number of bugs present in distributed systems [11]. Distributed systems are notoriously hard to debug and corner cases that rarely come up in practice often go untested. These bugs can lead nodes that are neither compromised nor corrupted to misbehave and propagate faulty updates to the rest of the system. This can cause catastrophic results and lead to extended outages of large portions of the system.

One of the motivations of this work is to address the issue of scalability in tolerating Byzantine faults in distributed systems. [1] describes an impossibility result which states that, to mask f byzantine failures in a distributed system requires $3f + 1$ servers. This limits the scalability of solutions that attempt to mask Byzantine faults. As opposed to trying to mask or prevent faults in the system, Byzantine detection [2] allows Byzantine faults to occur but provides eventual detection of these faults and guarantees this property even if only a single node in the system is correct. In this way, the system avoids this impossibility result

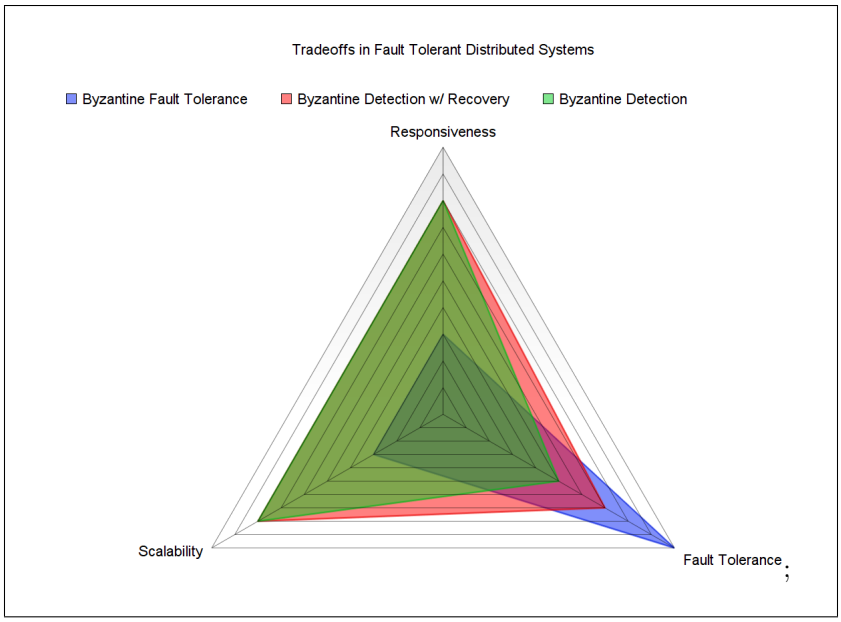


Figure 1.1: Tradeoffs present in fault tolerant distributed systems.

by not attempting to mask Byzantine failures proactively but rather to react to and recover from these failures. This provides a much more scalable means of dealing with Byzantine faults. Since the system only guarantees eventual detection, the protocol can run in the background with minimal impact on client latency. Additionally, it provides the property of being able to batch fault checking to periods of lighter system load to optimize for latency over fault tolerance in the common case. The Byzantine detection paper focuses on providing a system that detects faults and although it mentions some potential response strategies, it does not rigorously define or evaluate any of these strategies. Our work investigates this topic further by defining and evaluating a strategy to respond to detected failures. Figure 1.1 shows the tradeoffs present in fault tolerant system design and the potential gains from using our approach.

Though the Byzantine detector is defined to work with any state machine specification, we model our system for simplicity and without loss of generality as a replicated key-value store, in which clients can request either a *put* which updates the value of a variable, or a *get* which retrieves the value of a variable. For sake of argument, we also assume that each key is replicated at every node in the system. We define clients to be the entities that issue requests, and servers to be the entities that store keys and respond to requests. Our protocol requires that updates be signed with the private key of the sender, limiting potential client misbehavior only to *forking*, the scenario in which different updates are sent to different replicas. This scenario is explained further in Section 3.4.1.

This paper defines and evaluates a response strategy to detected failures. The goal of our strategy is to guarantee that, after a fork, our system will always converge to a consistent state through eliminating the

updates that caused the fork from the state of the server. Our system utilizes the Byzantine detector to detect the updates which caused a fork in the system, and defines a checkpoint-rollback strategy to allow replicas to eliminate faulty updates from their state, and allow the system as a whole to converge to a consistent state. Since the state of our system can be rolled back, it is possible for a client to read an older value after having previously read a newer value. We define this as a *lapse* and provide bounds on the number of lapses that can be experienced by a client through our lapse consistency model.

We evaluate our system to show that it is possible to not only detect but also respond to Byzantine faults in a more scalable manner.

Chapter 2

Related Work

2.1 Byzantine Fault Tolerance

Castro et al. [1] proved in their seminal paper on Byzantine Fault Tolerance the existence of a lower bound of $3f + 1$ on the number of servers required to mask f faults in a distributed system. They also propose a system which implements Byzantine fault tolerance in a more practical manner using the concepts of public key cryptography and digital certificates to authenticate updates in the system. It requires a 3-phase consensus protocol in which every node only commits if it is in agreement with the majority of servers. Our system attempts to address the high overhead and low fraction of fault tolerance of this approach by adopting Byzantine detection, allowing faults to be surfaced to the client, which allows for much reduced latency in the common case where no Byzantine faults are present.

Haeberlen et al. [2], [12] coined the notion of a Byzantine detector, which provides the foundation for this work. The Byzantine detector avoids the hard limits of Byzantine fault tolerance by not attempting to mask faults but rather attempting to detect faults after they occur. The work formalizes detectable failures and detectable ignorance of nodes and provides guarantees that faults will eventually be detected by requiring that all messages be digitally signed and exchanging evidence in the background of normal state machine operation. Since Byzantine detection does not attempt to mask faults, it allows for such optimizations as batching detection and evidence inspection to times of low activity on the system to give customers the best system performance. This flexible approach optimizes for the common case of no Byzantine faults but also provides a warning system if faults do occur. Our work extends the Byzantine detector to formalize and evaluate a method to react to faults and erase faulty updates from the state machine of the servers.

Distler et al. [13] discuss increasing the performance of Byzantine fault tolerant systems through not performing updates on all replicas but rather using a co-located selector component to replicate queries only at a subset of replicas. This method relaxes the consistency of the system while still ensuring Byzantine fault tolerance. Since this approach relies on BFT, it is limited to requiring $3f + 1$ servers to handle f faulty nodes, a limitation not present in our system. Our system is also orthogonal to the specification of the state

machine and can be used with data at any consistency level.

Mahajan et al. [14] discuss a Byzantine fault tolerant system that minimizes trust assumptions. It only requires that a single node in the system is correct in order to operate correctly. It does this in part through providing a new consistency model, Fork-Join consistency, to handle scenarios when malicious nodes cause a fork in the update history. Our system avoids the need to relax consistency under forks by maintaining checkpoints so that faulty updates in the system will be erased so that the servers converge to a consistent view of the data as if the faulty updates never happened.

Abd-El-Malek et al. [15] discuss an optimistic quorum based Query/Update protocol that provides Byzantine fault tolerance for less drastic performance reductions than other BFT systems. This system is an attempt to address the common case of non malicious failures, but still ends up suffering from the same drawbacks as it uses Byzantine fault tolerance offline, therefore requiring the number of correct nodes to double the number of faulty nodes.

2.2 Trusted Servers

BitTorrent [16] provides reliable storage and distribution of files as long as there is a trusted source to provide the metainfo file which contains the hash of each piece of each file being distributed. If the metainfo file is trusted, clients can reject any piece that does not hash to the same value as specified in the metainfo file. While this is a very efficient and reliable method of distributing files, this method does not apply to our model because we do not vet every update through a trusted server, but rather use trust assumptions only to enforce ordering. Our system is also agnostic to the specification of the state machine, so we do not restrict the distribution format of the protocol as BitTorrent does.

Veronese et al. [17] propose a system which uses trusted components in the system to reduce the required number of servers from $3f + 1$ to $2f + 1$ to tolerate f failures in the system as well as making the system simpler and more efficient. Compared to our system, Efficient Byzantine Fault Tolerance still requires an upper limit on the number of allowable failures since the system since the system is made to mask failures. It also requires a trusted component to maintain state in the system, whereas we only require a trusted component for ordering requests.

Chun et al. [18] explore the concept of a trusted log and relaxes liveness in order to provide safety when the number of faults in a system of N nodes is between $\lfloor \frac{N-1}{3} \rfloor$ and $2\lfloor \frac{N-1}{3} \rfloor$. It also provides an alternative approach that guarantees safety and liveness in the presence of up to half of the nodes in the system behaving arbitrarily faulty. It works by having a trusted computer base that maintains the history

of the system and can only append to its log to make it impossible for an adversary to fork the history of the system. This system is still restricted by hard limits due to attempting to mask Byzantine failures as well as trust assumptions which are not needed in our system except for request ordering.

Zhuo et al. [19] similarly use a write-once trusted table to reduce the overhead of machine fault masking.

Sen et al. [20] propose scalable algorithms for Byzantine fault tolerance with modest trust assumptions on a read-mostly workload. It does this by composing many small replica groups and adding small trusted channels to lessen the impact of Byzantine faulty nodes. By comparison, our system is agnostic to the workload and does not require any trust assumptions apart from request sequencing.

2.3 Corruption Tolerance

Behrens et al. [21] describe a system that handles data corruption without the need for replication, while solving issues that this posed in previous work, namely improving flat state, concurrency, and memory footprint issues.

Ateniese et al. [22] propose an efficient technique for checking if an untrusted server has a copy of the data without retrieving the data. This is a useful primitive in allowing users to audit what is being stored in the cloud on their behalf.

Chapter 3

System Model

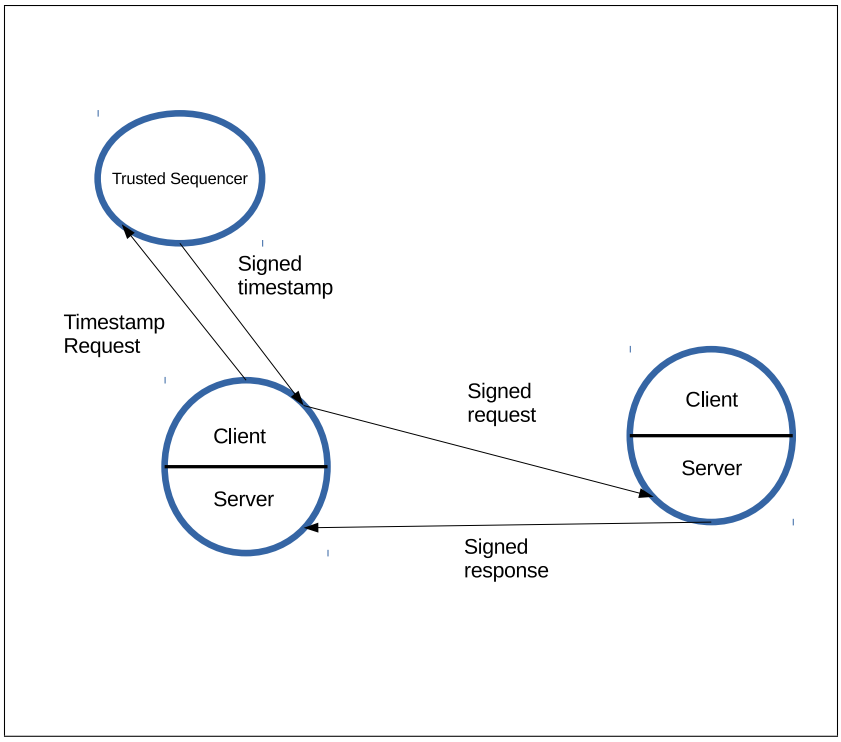
3.1 Overview

The system is organized as a replicated key-value store. Our ideas may be applied to any deterministic state machine but we will argue about the properties of the system using a key-value store for concreteness. In a replicated key-value store, each node i implements a key-value store K_i which stores key, value pairs on local storage and responds to two types of requests from other nodes in the system: *put* requests and *get* requests. *put* requests update the value of a key. They are structured as a tuple (k, v) where k represents the key to be updated, and v represents the value to which the corresponding key should be updated. *get* requests retrieve the value of a variable with a given key k.

In our system, each node may act as both a client and a server. It is necessary for the protocol to have clients and servers on the same machine due to the necessity for messages to be tagged with authenticators, a concept that will be addressed in Section 3.3.3. All messages sent are signed with the private key of the sender. Clients contact servers directly and can send individual updates to particular servers. All requests are logged at each server. All requests must also carry a timestamp signed by a trusted sequencer as described further in 3.2.3. For sake of argument, we also assume that each replica in the system replicates exactly the same set of keys. This is necessary for the log exchange that occurs in Byzantine detection described in 3.3.3. Figure 3.1 shows an overview of the system model.

3.2 Assumptions

Our strategy makes 3 key assumptions about the underlying system. Apart from these three assumptions, we allow nodes to fail in any manner including data corruption, malicious behaviour, and malicious collusion. These assumptions are similar to those of the Byzantine detector [2] since our work relies on the detection guarantees presented by the paper.



;

Figure 3.1: Overview of the system model

3.2.1 Determinism

We first require that the protocol in question is deterministic, meaning that the same inputs to the state machine A_i will always produce the same outputs. This is important to allow nodes to be able to perform conformance checks as previously described on the logs of other nodes. If the protocol were not deterministic, conformance checks could fail for correct logs as the same inputs do not necessarily always correspond to the same outputs.

3.2.2 Identity

We also make the assumption that each node has a public/private keypair and a unique identity to use in digitally signing messages and assume that the attacker will not be able to break the cryptographic signatures, meaning that no node can masquerade as another node by signing messages as them.

3.2.3 Total Ordering

Finally, we assume that requests will be totally ordered. For simplicity, we assume a trusted sequencer which can be queried to obtain signed timestamps that can be attached to messages. Clients contact the sequencer to obtain a timestamp signed by the sequencer before sending their requests. Requests with improperly signed

timestamps are dropped and the sender of such requests risks exposure. No traffic is routed through the sequencer, and the sequencer performs no other function except issuing timestamps. This trust assumption is relaxed in the Byzantine detector from [2] through the use of its commitment protocol, a form of consensus that provides the property that the sender of each message obtains verifiable evidence that the receiver logged the transmission. Relaxation of the total ordering constraint is an interesting and important problem but it is outside the scope of this paper and left for future work.

3.3 Byzantine Detector

The Byzantine detector [2] is a system that allows Byzantine faults to eventually be detected at every correct node in the system. Since our system assumes the availability of a Byantine detector, we formalize the description of the detector that is used in our model in this section.

3.3.1 Interface

The Byzantine detector defines two types of failure indications that it may send to the application process.

- $suspected_j$ indicates that there is evidence that node j may be ignoring certain inputs
- $exposed_j$ indicates there exists a proof that node j is faulty

3.3.2 Guarantees

The Byzantine detector [2] defines two classes of faults: detectable faultiness and detectable ignorance. A replica is said to be detectably faulty if either it has sent an invalid message, such as an improperly signed message, or has sent a divergent update. A node is said to be detectably ignorant if it is not detectably faulty and there is evidence that it is ignoring some inputs. The Byzantine detector then provides the following guarantees:

- Eventually, every detectably ignorant node is suspected forever by a correct node
- If a node i is detectably faulty with respect to a message m, then eventually some faulty accomplice of i is exposed or forever suspected by every node
- No correct node is forever suspected by a correct node
- No correct node is ever exposed by a correct node

3.3.3 Protocol

The Byzantine detector requires every node in the system to maintain a log of all of the inputs and outputs to its state machine A_i. Nodes must frequently publish a signed hash of this log, called an authenticator, to other nodes for verification. Nodes can then forward authenticators to propagate them through the system. Nodes must digitally sign all messages sent and acknowledge all messages received.

Auditing

Each node i can periodically be chosen for an audit by every other node. During the audit, each auditor asks the node for a signed copy of the log up to the point of the previous audit. The auditor can then validate the log against the most current authenticator it has received during the normal operation of the protocol. Should the node i being audited refuse to respond to the audit, it will be suspected by the other nodes in the system.

After verifying the log against the most recent authenticator, the auditor proceeds to check if the log matches all previously received authenticators. This process is called a consistency check. If this fails, the auditor now has a proof that can be used to expose node i.

In addition to the consistency check, an auditor may also perform a conformance check. During a conformance check, the auditor creates a local copy of the state machine and applies the input from the log of i to make sure that the inputs in the log of node i actually produce the same outputs.

Evidence

Upon receiving or producing evidence of a faulty update by a node in the system, a correct node can evict the node that originated the update since it is provably faulty. Evidence of faults is propagated by every correct node to all other nodes to ensure that every correct node is able to detect every fault. Each node can then independently verify the evidence based on its log and its history of past authenticators. Since the system only provides eventual guarantees, evidence verification can be batched for times of light system load to optimize for latency in the common case.

3.4 Failure Scenarios

Byzantine failures encompass all possible failures including crashes, data corruption, malicious nodes, and colluding malicious nodes. We enumerate in this section which types of these failures are possible under our system model and assumptions.

3.4.1 Client

Since all requests in the system are signed with a node's unique private key, many potential failure scenarios involving nodes masquerading as other nodes are impossible in the system under the assumptions outlined in Section 3.2. If a node sends improperly signed data, each correct receiving node will drop the update. This means that the only possible malicious failure scenario, is that of *forking*.

Forking is a scenario in which a malicious client sends different updates to different replicas. This is possible since clients send updates individually to each server as described in Section 3. This would cause replicas of the same key to hold different values after each replica receives the update. In this situation, the data held at different replicas is inconsistent, and will never converge to a consistent state in the absence of a Byzantine recovery strategy. We define a consistent state to be one in which the values held by all replicas are identical for every key stored. We define a *fork* to be a state of the system in which all replicas have received all updates, yet there exists a pair of replicas of a key that hold different values. We additionally define a *divergent* update to be an update that causes a fork in the system. Section 4 describes how our recovery strategy allows the system to recover from a fork.

Since requests in the system are totally ordered, a client may attempt to place an improper sequence number on its update in order to have updates delivered out of order. Our protocol disallows this by requiring that requests be ordered by a trusted sequencer or by consensus as described in Section 3.2.3. Since sequence numbers must be signed by the trusted sequencer, they are unforgeable by malicious processes under the assumptions of Section 3.2.

In absence of forking, which means that clients send properly signed data to all replicas, malicious failures are not possible since any fault would be indistinguishable from a correct update.

3.4.2 Server

There are multiple potential failure scenarios involving the server.

A server may fail to respond to all updates. In this case, it will be indistinguishable from a crashed node, and this failure scenario can be handled using traditional crash tolerance methods, namely replication in our system.

A server may selectively respond to some updates and not respond to other updates. In this case, the server will eventually be suspected forever by some correct node. Since no server state in our key-value store is changed as a result of replies from servers, no rollback is necessary in the case of this failure. Other response strategies including evicting the faulty server can be employed.

A server may send false data in response to a *get* request. In this case, the Byzantine detector from

Section 3.3 would eventually expose the faulty server since an audit of the faulty server's log would fail a conformance check. Since this scenario does not impact the stored state of the server, no rollback is necessary as a result of this failure scenario. The faulty server may, however, be evicted as a result of exposure.

Chapter 4

Core Protocol

4.1 Overview

The overall goal of our strategy is to eliminate from the state of every replica all updates that fork the system by rolling back the state of servers to a consistent state before the fork. When an update is exposed as faulty by the Byzantine detector, servers replay all updates from their logs before those updates exposed as faulty by the Byzantine detector. Since every node will detect the faulty update based on the properties of the Byzantine detector described in Section 3.3, the value of a key at every replica will eventually be the same. Our system guarantees that every divergent update will eventually be erased from the state of every server, and that the values held by replicas for each key will eventually converge to a consistent state.

4.2 System Architecture

Each node i in the system implements two key-value stores A_i and C_i in addition to a Byzantine Detector B_i as described in Section 3.3. A_i is the active key-value store that stores the current state of the system and responds to client requests. C_i is a key-value store used for storing a *checkpoint* of the system which contains old updates whose log entries were discarded. This helps mitigate the cost of rolling back the system in the event of a faulty update being detected. C_i does not respond to client requests and can only be updated by the Byzantine detector when it discards log entries. The motivation and characterization for this approach to checkpointing is described further in Section 4.4. Figure 4.1 gives an overview of the system architecture.

4.3 Asynchronous Log Replay

When an update is exposed by the Byzantine detector, the node i creates a temporary key-value store T_i to facilitate the rollback of system state. T_i is initialized to be a copy of the checkpoint key-value store C_i if it exists. The purpose of the checkpoint and how it is created is described in Section 4.4. The node then starts a separate thread of execution which sequentially applies each update from its log to T_i before the

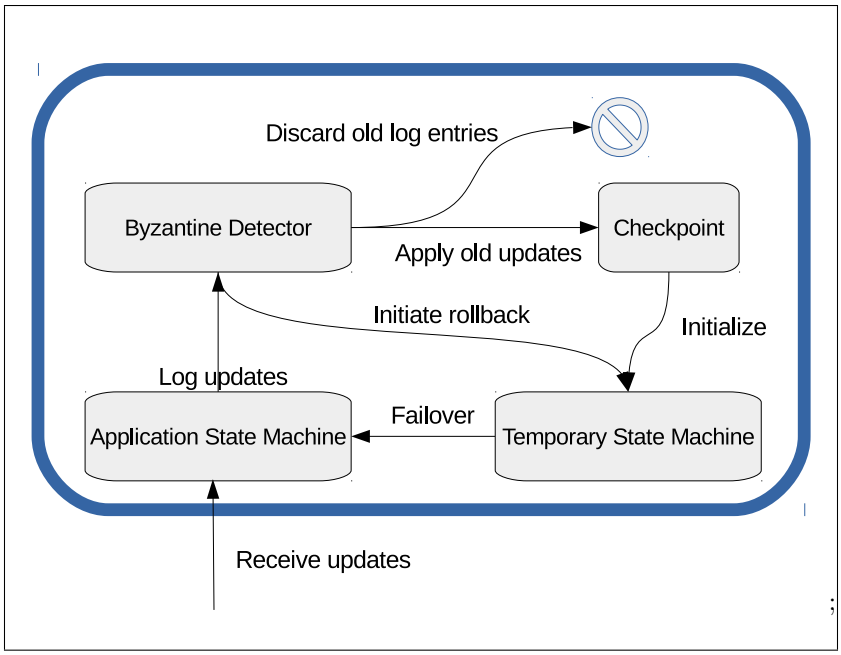


Figure 4.1: Recovery architecture of a single replica

entry that is exposed as faulty. The server makes T_i its active key-value store once all updates before the divergent update have been replayed from the log. For requests that happen concurrently with rollback, i.e. after the fork, we choose to discard these updates in favor of preserving a correct system state. The reason for choosing this is that there are often dependencies in the pattern of writes to the system, meaning that applications often read state before writing. With this in mind, updates that are dependent on the faulty system state after the fork may not be trustworthy and it would not make sense to apply updates that were potentially conditioned on that faulty state. Algorithm 1 summarizes the rollback procedure.

Algorithm 1 On exposure of a faulty update m at correct node i

1: Create a new state machine instance T_i
2: Initialize T_i to C_i if it exists
3: Start a new thread of execution
4: Sequentially replay all updates in the log except for m to C_i
5: Replace live state machine A_i with cleansed state machine T_i

4.4 Checkpointing

Our strategy allows for checkpointing to reduce the number of updates that must be replayed from the log during rollback. Since there is no defined time when a node can positively say that an update in its log is fault-free due to the eventual guarantees provided by the Byzantine detector, a node cannot simply create a

checkpoint of validated updates to which it can roll back. The Byzantine detector [2] allows for discarding log entries older than some time T_{trunc} to save space on each node. Our system takes advantage of log pruning to provide a point to which the system can roll back. Each discarded log entry is applied to a checkpoint key-value store C_i. This checkpoint is used to initialize the state of the temporary state machine T_i during asynchronous rollback as described in 4.3. This provides a way to preserve the state of updates which no longer appear in the log. Algorithm 2 summarizes the checkpointing procedure. This also bounds the number of updates that need to be re-applied upon rollback to the number of updates applied since the last round of pruning.

With respect to storage size, there are two reasons for maintaining a separate key-value store to hold the state of old updates instead of simply maintaining the log entries for those updates. First, storing a separate key-value store means that *get* requests no longer contribute to storage size since only a flat state is stored. Secondly, *put* requests for the same variable are flattened to only include the latest value for that variable since the individual requests are no longer stored. These two reasons make the checkpoint more attractive than maintaining complete logs for old updates.

Algorithm 2 On discarding of a log entry e

1: Apply the operation specified in e to C_i
2: Discard e from the log

4.5 Consistency

Since we are not performing Byzantine fault tolerance, we necessarily have to expose faults to clients to avoid the impossibility result of [1], meaning that clients may observe an inconsistent view of the data. To motivate this, consider a replicated key-value store in which a malicious client attempts to create a fork in the system by requesting that the value of variable x with initial value 0 be updated to 1 on one replica and to 2 on another. In a Byzantine fault tolerant system, this update would not be performed as the replicas would not be able to come to an agreement on the value to which x should be updated. All replicas would still hold the original value of 0 and in this way the fault is masked from clients. However, in the case of a non-Byzantine fault tolerant system, the update would be performed. This means that subsequent reads of x will either return 1 or 2 depending on which replica is accessed and the system can never converge to a consistent state.

Our system guarantees that the system will eventually be able to converge to a consistent state after a fork. This guarantee is formalized in Section 4.6.3. In the previous example, since each node would be able

to expose the divergent update through its Byzantine detector, each node would roll back its state to the time before the fork, and all replicas of x would converge to a value of 0.

Since the system uses checkpointing to roll back its state to one that existed prior to the fork, a client issuing a *get* request may observe an older value than it had previously read on the same object. We call this a consistency *lapse*. Our strategy bounds the number of lapses that a client can observe to at most one consistency lapse per faulty update. We call this guarantee lapse consistency. A client issuing *get* requests will observe strong consistency until a Byzantine fault occurs. At this point in time, one of two scenarios will happen. Either the client receives conflicting replies from the replicas that it queried (an inconsistency), which would be the case if a divergent update was issued and not yet detected, or the client observes a lapse. Our system provides that eventually *get* requests will no longer return inconsistent values, but a client may observe at most one lapse per faulty update. We prove our consistency guarantees in Section 4.6.3.

4.6 Guarantees

We partition the guarantees of our system into four categories: Completeness, Accuracy, Consistency, and Liveness.

4.6.1 Completeness

Our system guarantees *eventual completeness*. That is, every divergent update will eventually be erased from the state of all correct servers.

Proof

We first note that the Byzantine detector guarantees that if a node i is detectably faulty with respect to a message m, then eventually some faulty accomplice of i is exposed or forever suspected by every node.

In the case of a divergent update m, the client i issuing the update is detectably faulty with respect to m since i signs both versions of the update and each version has the same unforgeable timestamp. These two updates form a proof of the faultiness of i which will be exposed during the authenticator exchange between the two nodes receiving the different versions of the update. Since i sends its messages directly to the other replicas, there are no accomplices in this case, meaning i itself will be exposed.

When i is exposed, this exposure will contain a signed log whose hash does not match a previously received authenticator. This proof will be spread to all replicas, and eventually the Byzantine detector of each replica will expose i and subsequently roll back its state to its state before applying m. Since m will

eventually be exposed at every correct server, and every exposure leads to a rollback that eliminates m from its state, m will eventually be erased from all servers.

■

4.6.2 Accuracy

Our system guarantees that no correct update will be erased from the state of any correct server.

Proof

Since the Byzantine detector as described in 3.3 guarantees that no correct node will be exposed, and that rollback only occurs on exposure of a faulty update, no rollback will occur for a correct update. Therefore, a correct update cannot be erased from the state of a correct server. ■

4.6.3 Consistency

In this section, we will prove the following claims:

1. Eventually after a fork, *get* requests will receive consistent replies from all correct replicas
2. A client may observe at most one consistency lapse per faulty update

Proof

For claim 1, we utilize the fact that the system provides eventual completeness as proven in Section 4.6.1. This means that every correct node will eventually erase from its state the effects of every divergent update. Therefore, the state of every correct node will eventually reflect only correct updates. Since requests are totally ordered, all servers will have processed the same set of *put* requests before processing a *get* request. This means that every correct server responding to a *get* request has processed the same set of correct *put* requests. Therefore, eventually all correct servers responding to a *get* request will respond with the same value. ■

For claim 2, let us denote the order of read operations on an object o at a client c as a sequence $r_1^c(o), r_2^c(o), ..., r_n^c(o)$ where $r_i^c(o)$ is the ith read operation performed at c on o. Let $t(r_i^c(o))$ denote the timestamp of the ith read operation at c on o. Formally, we define a consistency lapse as a set of read operations $r_i^c(o)$, and $r_j^c(o)$ where $j > i$ such that $t(r_j^c(o)) < t(r_i^c(o))$. For purposes of demonstration, suppose the contrary. That is, suppose that the client observes $l > 1$ consistency lapses for a single faulty update. This means that, in a set of l read operations $r_{i_1}^c(o), r_{i_2}^c(o), ..., r_{i_l}^c(o)$ where $i_1 > 0$ and $i_l > i_{l-1} > ... > i_2 > i_1$

such that $t(r^c_{i_l}(o)) < t(r^c_{i_l-1}(o)), t(r^c_{i_{l-1}}(o) < t(r^c_{i_{l-1}-1}(o), ..., t(r^c_{i_1}(o) < t(r^c_{i_1-1}(o)$. The client can observe $t(r^c_j(o)) < t(r^c_i(o))$ where $j > i$ only if the system has rolled back due to a faulty update since servers support no other operation to eliminate an update that it has previously seen. This means, to observe $t(r^c_{i_l}(o)) < t(r^c_{i_l-1}(o)), t(r^c_{i_{l-1}}(o) < t(r^c_{i_{l-1}-1}(o), ..., t(r^c_{i_1}(o) < t(r^c_{i_1-1}(o)$, the system must have rolled back l times for a single faulty update. This is impossible since each replica rolls back only after verifying a new piece of evidence of the exposure of an update which occurs at most once per faulty update. We have arrived at a contradiction and shown that a client may observe at most one consistency lapse per faulty update. ■

4.6.4 Liveness

There is no deadlock possible with our strategy.

Proof

The necessary conditions for deadlock are as follows:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular Wait

We argue that our system violates condition 2, making deadlock impossible. Since rollbacks happen in a separate thread of execution, the active key-value store at a node can continue servicing requests the entire time without waiting. Since the log is shared between the active state machine and the state machine which is rolling back, we use mutual exclusion to synchronize access to the log. The only work that is done while the lock is held is to either append an element to the log or read an element from the log and release the lock. Since there is no hold and wait in our system, this violates one of the necessary conditions for deadlock, making deadlock impossible in our strategy. ■

Chapter 5

Alternative Approach

The approach presented in the previous chapter offers recovery from Byzantine failures using a Byzantine detector [2]. In this section, we explore an alternative approach and contrast the tradeoffs of this approach with the approach mentioned in the previous chapter.

5.1 Redundant Systems

Instead of requiring servers to implement a Byzantine detector, this approach requires two separate sets of servers to replicate keys. One set of servers admits requests from clients using traditional crash fault tolerant (CFT) consensus to order requests, while a separate set of servers running in parallel runs Byzantine fault tolerant (BFT) consensus. This system optimizes for performance in the common case of no Byzantine failures by only requiring the client to wait for responses from the crash tolerant cluster but also provides the property that any fault can eventually be recovered from since there always exists servers which hold the correct state.

5.1.1 Architecture

The system is organized in a client-server model. Clients issue requests to a set of servers which run crash tolerant consensus. This could be implemented either by the client contacting a leader of the crash tolerant group or contacting each of the individual servers directly. Servers implement a key-value store as described in Section 3. Servers in the CFT system forward all received requests and their responses to the Byzantine fault tolerant cluster. Clients forward received responses to the Byzantine fault tolerant servers. There are $3f + 1$ nodes in the Byzantine fault tolerance cluster, in order to tolerate up to f Byzantine failures. The Byzantine fault tolerant servers will always maintain a correct system state to which the crash tolerant servers can reset in case of failure.

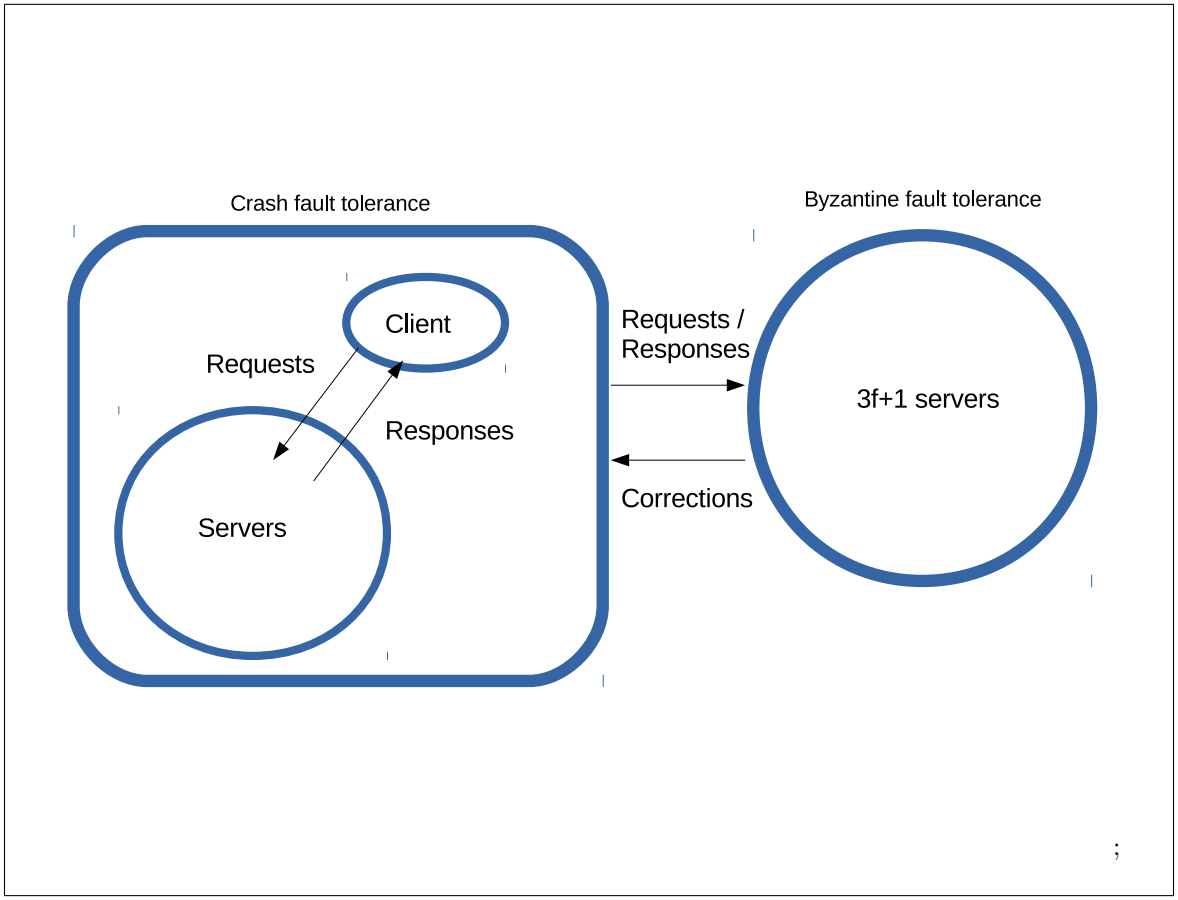


Figure 5.1: Architecture of redundant systems approach

5.1.2 Operation

A client issues either a *put* or *get* request to the servers in the crash tolerant cluster. The crash tolerant cluster uses consensus to assign a timestamp to the request, applies the request in order, and replies to the client. A client issuing a *get* request may either wait for a single reply to optimize for speed, or wait for $f+1$ matching replies to be guaranteed that the reply reflects the current state at least one correct server. This state may or may not be faulty depending on how frequently servers in the crash tolerant cluster exchange their state with the Byzantine fault tolerant cluster. The CFT servers forward both the requests they receive and the responses they issue to the Byzantine fault tolerant cluster along with the order in which the requests were applied. Clients forward received replies to the Byzantine fault tolerant cluster. Upon receiving the requests and their order, the Byzantine fault tolerant cluster can then apply the request in a Byzantine fault tolerant manner.

The set of nodes running the crash tolerant version of the state machine can fail over to the state of the Byzantine fault tolerant set of nodes to recover from any Byzantine failure that occurs in the system. Servers can achieve this by querying the Byzantine fault tolerant cluster about the value of each stored key. Since there can be up to f faulty nodes in each cluster, each server would need to collect $2f+1$ replies from the Byzantine fault tolerant cluster about the value of each key. Once these replies have been collected, the value agreed upon by at least $f+1$ servers is guaranteed to be the correct value and the server can repair its state using that value.

5.1.3 Comparison

Since the normal operation of requests is unhindered by the system as opposed to each node having to implement a Byzantine detector, this approach provides less latency for clients in the common case. This system, however, is quite expensive compared with the checkpointing approach described in Chapter 4 due to the need for the more servers than the checkpointing approach.

Chapter 6

Implementation

We implemented the system in C++ using ZeroMQ to handle message queues and protocol buffers as the wire format for all communications. We use libgcrypt for all cryptographic functions which includes hashing and signing data. We use RSA for our public key cryptography scheme. Each server has a unique id number and a unique public-private key pair. The public keys are known to all replicas while private keys are available only to the servers that originally generated them. All messages include a digital signature signed with the private key, an authenticator which is a signed sha256 hash of the log that each replica maintains, as well as the id number of the requesting server so that the receiving server knows which public key to use to verify the message. Each server also acts as a client, and can issue requests signed with the same private key that it uses to sign responses.

Our implementation simulates the rollback approach mentioned in Chapter 4 without the checkpointing optimization. Every time rollback occurs, the log must be replayed from the beginning to roll back the system to the correct state. From the byzantine detector, our simulation supports authenticator propagation, histories of past authenticators, auditing, and evidence verification. For the purposes of our experiments we seek to isolate the overhead of the rollback strategy, so we simulate immediate detection of faulty updates by implementing a malicious client that simply attaches improperly signed authenticators to its messages based on a failure probability, which allows each node to instantly detect the Byzantine fault.

We implement a replicated key-value store as described in Chapter 3. We use a hash map at each replica to associate keys with values. The key value store supports two operations: $put(key, value)$ which inserts the provided key into the store with the provided value as well as $get(key)$ which returns the value associated with the key. Our system replicates data at all servers present in the configuration. This is not necessary for our protocol, but it simplifies the system model and makes it easier to reason about our results.

Each node responds to three types of requests: *put* requests, *get* requests, and *audit* requests. *put* requests contain the name of the key as well as the value to which the key should be set. After verifying a *put* request, each replica applies it to its local key value store, and replies to the client with a simple acknowledgement. *get* requests contain the name of a variable. Replicas respond to *get* requests with the

value of the variable in their local key value store. *audit* requests contain an upper and lower timestamp, and replicas reply to these requests by sending all log entries whose timestamp falls between these two entries.

Chapter 7

Evaluation

We evaluate our system to show the overhead of our approach and argue that the system still preserves scalability and responsiveness while tolerating Byzantine faults.

7.1 Experimental Setup

For our evaluation, we use up to 9 instances of Digital Ocean [23] servers in the NY2 region, each with 512MB of RAM, 20GB of disk space, and a single CPU. Each server runs an instance of the protocol and the daemon is built and deployed through scripts using ssh. Each server generates traces in JSON format of all messages sent and received as well as peak memory usage. These traces are copied to a local drive where they are then aggregated and analyzed through a separate suite of custom programs.

For each of the experiments, there is a single client issuing a sequence of *put* requests one after the other without waiting for replies. This client is connected to other nodes in the system whose servers process requests and reply to the requesting node. All request times and corresponding reply times are traced in order to measure request latency. Since each request gets sent to each of the replicas in the system, we measure latency as the time of the last received reply minus the time of the first request issued. We additionally measure peak memory usage at each node and total network usage of the experiment. Our experiments were each run with a single repetition except for the low failure rate tests which were done with three repetitions.

7.2 Overview

Overviews of the various runs for extreme cases (highest and lowest parameter values) are shown in in figures 7.3, 7.1, and 7.2. $faultRate$ represents the probability of a Byzantine failure occuring for a given request as described in Section 7.1.

We model a Byzantine failure in our experiments as an improperly signed authenticator, allowing each node in the system to immediately recognize the update as faulty. This means that we make the assumption

that detection is immediate.

numRequests represents the number of requests issued by a single client in the experiment. *payloadSize* is the number of characters in the value field of each put request. *numReplicas* represents the number of replicas used in the experiment. We abbreviate our model as BFR for Byzantine Fault Recovery and compare it against the normal implementation which has neither Byzantine detection nor recovery capabilities and hold all other parameters the same. The parameters of our overview experiments are shown in Table 7.1. These measurements give a good idea of which variables have the most effect on each of the metrics we studied: latency, total network usage, and peak memory usage. Latency is measured as the time between the client request, and the receipt of the last reply from the set of replicas. Total network usage is the sum of sizes of every message sent in the system over an entire run of the experiment. Peak memory usage represents the most memory used at any point in time during a run of an experiment.

To measure overhead in our experiments, we run each experiment using both the normal implementation and the BFR implementation. We then divide the outcome of the BFR experiment by the outcome of the normal experiment. We then multiply by 100 and subtract 100 to compute the percentage overhead. This means that if a system were compared with itself, it would have 0% overhead. This provides us with a normalized figure of how much our system impacts the overhead of each of these metrics.

Table 7.1: Run parameters for overview experiments

numRequests	payloadSize (# of characters)	numReplicas	faultRate
100	1	9	0
100	1	2	0
100	10000	9	0
100	10000	2	0
100	1	9	0.9
100	1	2	0.9
100	10000	9	0.9
100	10000	2	0.9

7.2.1 Latency

Figure 7.1 shows the effects of fault rate, number of replicas, and payload size on the average request latency. We find that the number of replicas has the biggest effect on latency since the client waits for the slowest replica for each request. This is evaluated further in 7.4.1. Increasing the fault rate surprisingly decreases the average latency in the system for a variety of reasons described in section 7.5.1.

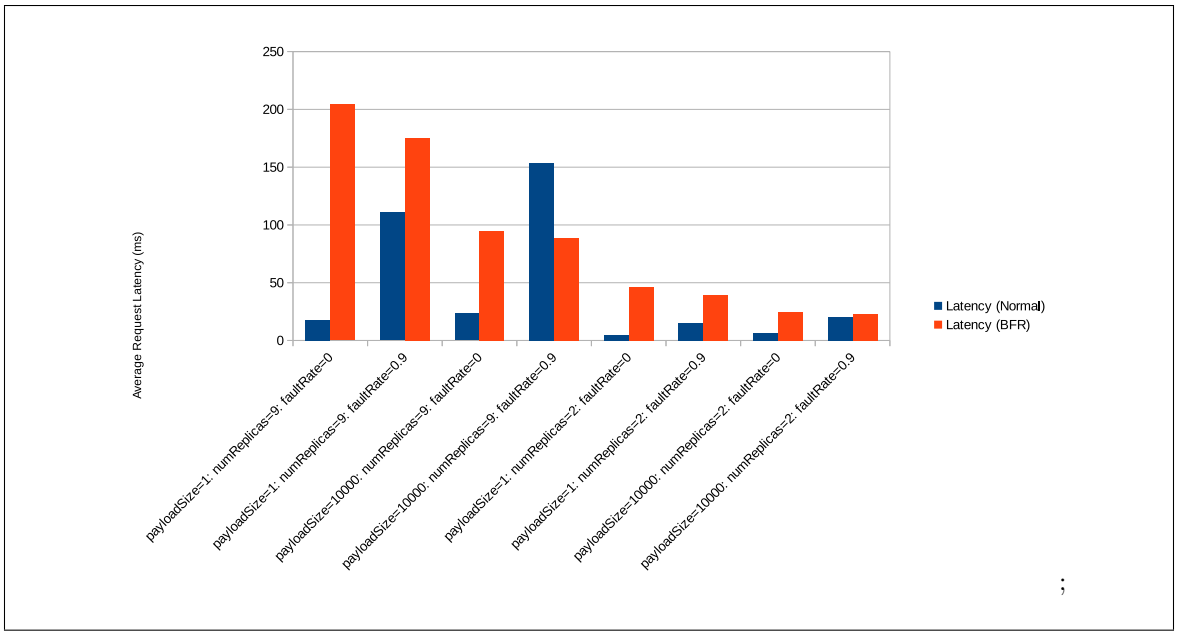


Figure 7.1: Overview of latency usage for various run parameter configurations

7.2.2 Total Network Usage

As is visible in Figure 7.2, the two variables with the largest impact on total network usage are the size of the payload in each request, and the number of replicas in the system. As the number of replicas increases, requests need to be sent to each of these replicas, which consumes a proportional amount of network bandwidth. Similarly, as payload size increases, each request becomes larger and therefore consumes more network bandwidth. This is described further in section 7.4.3. Introducing Byzantine failures into the system causes total network usage to increase due to replicas having to forward evidence to the other replicas in the system. However, due to replicas not replying to faulty updates, the network usage is much reduced compared to the no failure case. This is described in more detail in section 7.5.3.

7.2.3 Peak Memory Usage

Figure 7.3 shows that the number of replicas has the biggest effect on peak memory usage. This is primarily because each node has to store authenticators received by all other replicas, which consumes memory proportional to the number of replicas in the system. This is described further in secion 7.4.2. The failure rate in the system is inversely proportional to the peak memory usage because the system rolls back after each faulty update, meaning that it only has to store the correct updates. This means that as the failure rate increases, less updates need to be stored. This is described further in section 7.5.2.

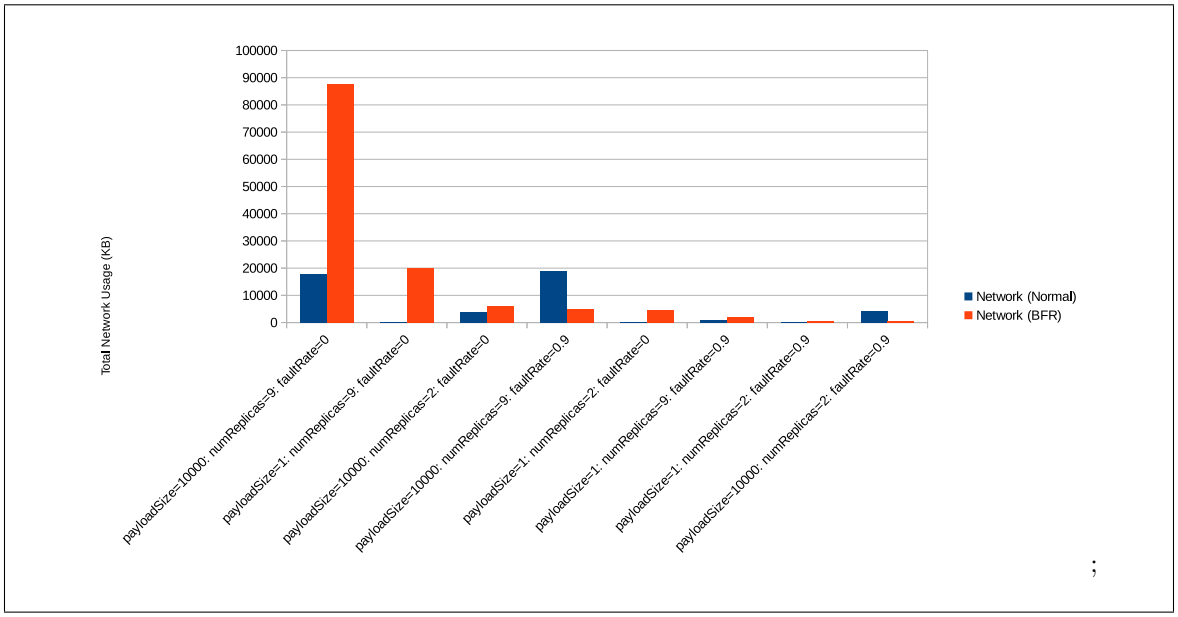


Figure 7.2: Overview of total network usage for various run parameter configurations

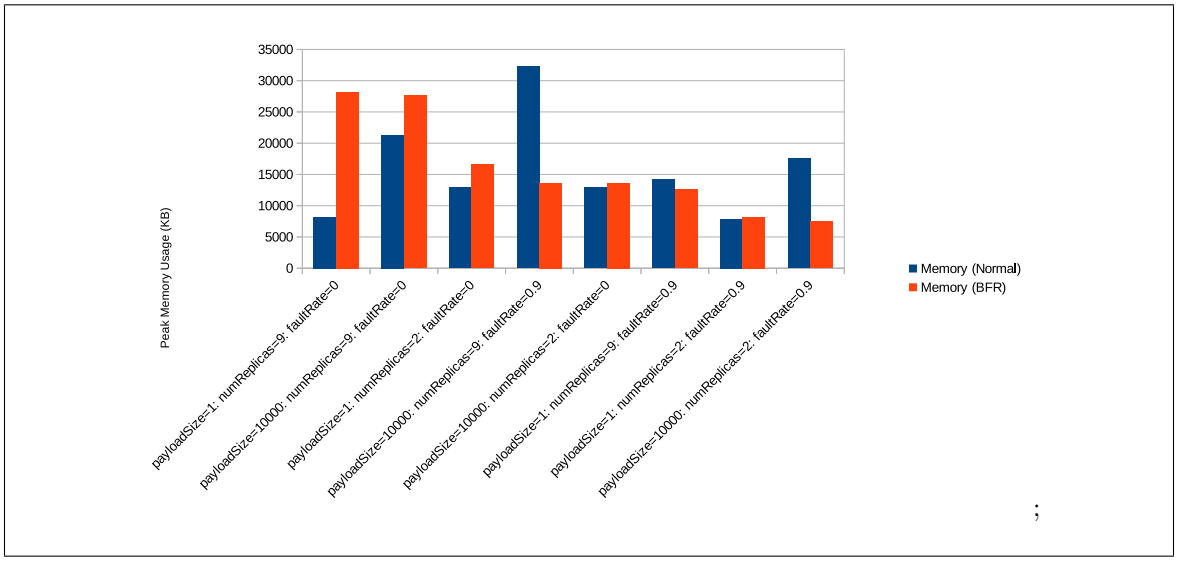


Figure 7.3: Overview of memory usage for various run parameter configurations

7.3 Payload Size

Our first observation is that while the network usage increases with the payload size, the total network usage *overhead* of our protocol diminishes as the size of each payload increases. We observe a high network usage overhead at small payload sizes since the size of the digital signature dominates the size of the request. However, as the size of requests increase, the overhead of our protocol diminishes as digital signatures no longer dominate the size of the request. From this we conclude that, from a network usage standpoint, our protocol requires less overhead for larger request sizes and more overhead for smaller request sizes. A potential solution to improve overhead of small requests is to batch small requests, but batching is not addressed in our work.

Table 7.2: Parameters for payload size experiment

numRequests	payloadSize (# of characters)	numReplicas	faultRate
100	1	2	0
100	10	2	0
100	100	2	0
100	1000	2	0
100	10000	2	0

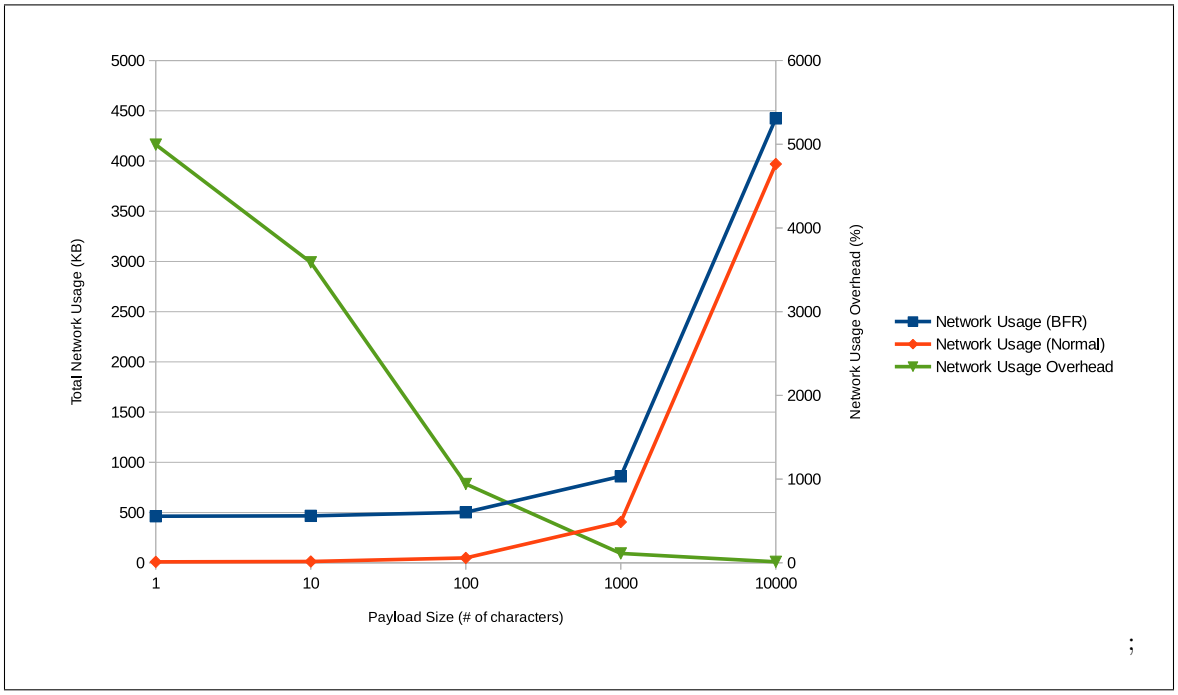


Figure 7.4: The effect of payload size on total network usage

7.4 Scalability

We next evaluate the scalability of our approach by tracking the overhead for various metrics while varying the number of replicas in the system. As in previous experiments, overhead is measured by dividing the result of each run with byzantine recovery turned on by the corresponding result of running the non-Byzantine version of the system. This normalizes the data to isolate only the additional overhead of running Byzantine recovery over a non-Byzantine implementation. In each of the following experiments, the payload size was fixed to 10000 characters and no Byzantine faults were introduced into the system.

Table 7.3: Parameters for scalability testing

numRequests	payloadSize (# of characters)	numReplicas	faultRate
100	10000	2	0
100	10000	3	0
100	10000	4	0
100	10000	5	0
100	10000	6	0
100	10000	7	0
100	10000	8	0
100	10000	9	0

7.4.1 Latency

Figure 7.5 evaluates the latency overhead as a function of the number of replicas in the system. As can be seen, although the latency overhead is a high constant, the latency overhead tends to not be affected by the number of replicas in the system. Since digital signature computation is the operation that affects latency the most in the BFR system, and this computation is done in parallel at every server processing a request, the overhead of latency is constant when compared with the normal implementation. In addition, the actual latency of both implementations increases linearly with the number of replicas in the system since clients wait for replies from all replicas and is therefore limited by the speed of the slowest replica.

The largest component of the latency overhead is the necessity to compute digital signatures for every request in addition to the high amount of background authenticator traffic that is present in the system that can hold up message queues. The rate of authenticator traffic can be mitigated by reducing the rate at which authenticators are broadcast or even batching all authenticator propagation and evidence checking to times of low load on the system which would leave only the cost of computing authenticators on outgoing messages to account for the added latency.

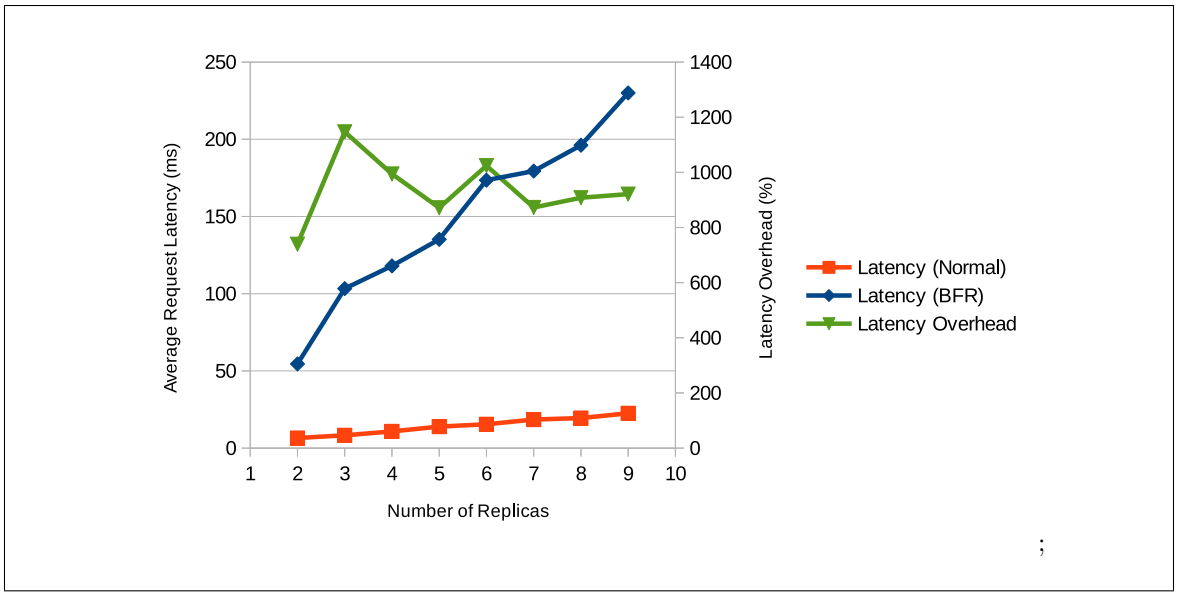


Figure 7.5: Latency overhead as a function of the number of replicas

7.4.2 Memory Overhead

As shown in Figure 7.6, the memory overhead shows slow linear growth as the number of replicas in the system is increased. This is due to the need to store past authenticators in order to achieve Byzantine detection. Authenticators are produced with every request by every node in the system. Storing authenticators is important for Byzantine detection in being able to expose a node based on the history of updates they sent. In addition, the peak memory usage of both implementations also grows linearly as, in both cases, each replica must log every request made to the system as well as store keys and values in its local hash map. Each replica must also store membership information about the other replicas in the system to be able to connect to them, which increases proportionally to the number of nodes in the system.

7.4.3 Total Network Usage Overhead

Figure 7.7 shows that the total network usage overhead of the protocol is unaffected by the number of nodes in the system. This shows that Byzantine fault recovery incurs only a constant level of overhead in terms of network usage that does not increase with the number of nodes. This is due to the fact that the only extra network bandwidth used by the system in non-failure scenarios is the extra space required for the digital signatures and authenticators on each message sent.

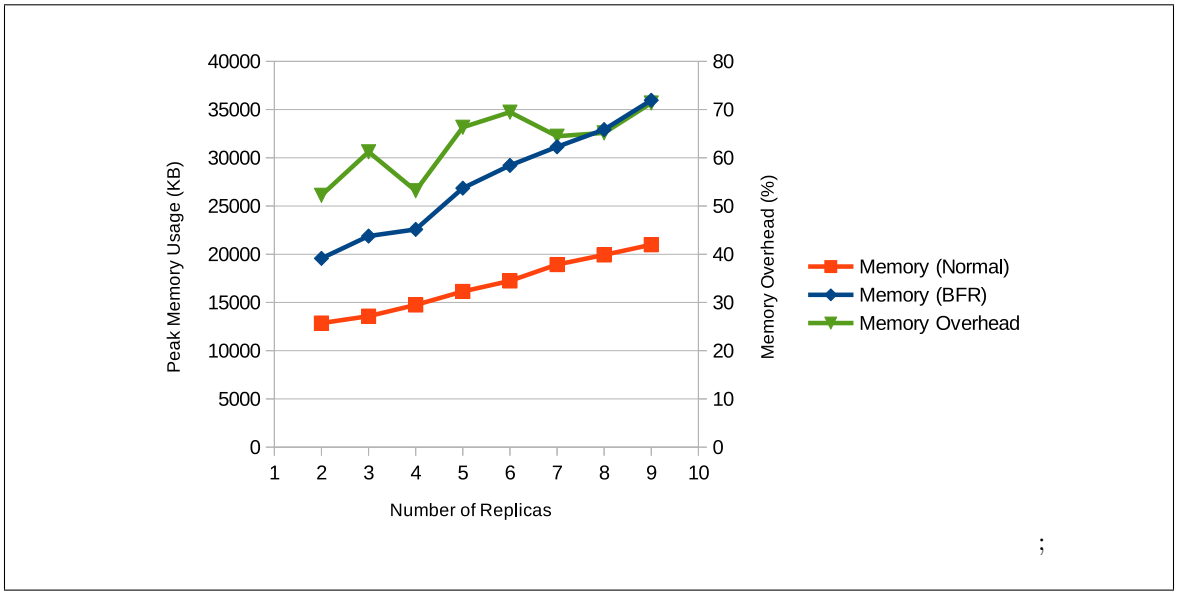


Figure 7.6: Peak memory usage overhead as a function of the number of replicas

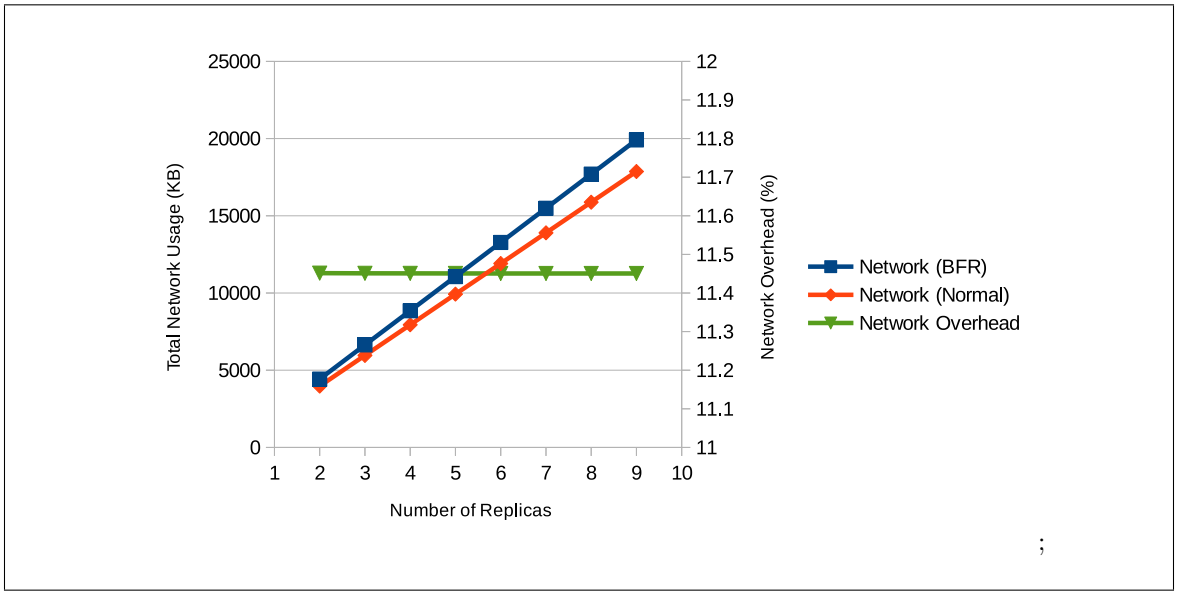


Figure 7.7: Total network usage overhead as a function of the number of replicas

7.5 Fault Tolerance

To show the performance of our protocol, it is important for us to evaluate the system's overhead during failure scenarios. Each of the following experiments was run with a failure rate, which is a probability from 0 to 1 which represents how likely a node is to send an invalid authenticator with each request, from which each replica can independently detect and recover. We evaluate the system by varying this failure rate and reporting five metrics: latency, memory, and total network usage overhead which were studied in the scalability section during the no failure case, as well as the number and duration of rollbacks that occur due to Byzantine faults. We run two sets of experiments one with a widely varing failure rate to show the overall, and one focusing specifically on smaller failure rates, to model the property that Byzantine failures are often rare in practice.

Table 7.4: Parameters for overall fault tolerance testing

numRequests	payloadSize (# of characters)	numReplicas	faultRate
100	10000	5	0
100	10000	5	0.1
100	10000	5	0.2
100	10000	5	0.3
100	10000	5	0.4
100	10000	5	0.5
100	10000	5	0.6
100	10000	5	0.7
100	10000	5	0.8
100	10000	5	0.9

Table 7.5: Parameters for fault tolerance testing under low failure rates

numRequests	payloadSize (# of characters)	numReplicas	failureRate
10000	100	4	0
10000	100	4	0.0001
10000	100	4	0.001
10000	100	4	0.01
10000	100	4	0.1

7.5.1 Latency

Figure 7.8 shows the effect of Byzantine failures on latency in the system. This was a surprising and unexpected result but can be attributed to a variety of reasons. The biggest reason that latency decreases as failure rate increases is that there is less network congestion due to message replies since nodes do not

reply to faulty updates. This, in turn, means that there are less messages in message queues to handle replies, so there is a greater chance that an actual request will be processed immediately. Additionally, since replicas store fewer keys under high failure rates due to repeated rollback, there are fewer entries in the local key-value store, meaning less chance for collisions in the hash table which leads to faster lookup times. Since the normal implementation does not react to Byzantine failures, it continues to amass all of the previous updates it has seen, which increases latency in a slow linear growth for the reasons mentioned previously. This leads to a convergence of latency performance across the two implementations, meaning that overhead is greatly decreased as the failure rate increases. Figure 7.9 which examines latency over lower failure rates shows a similar trend.

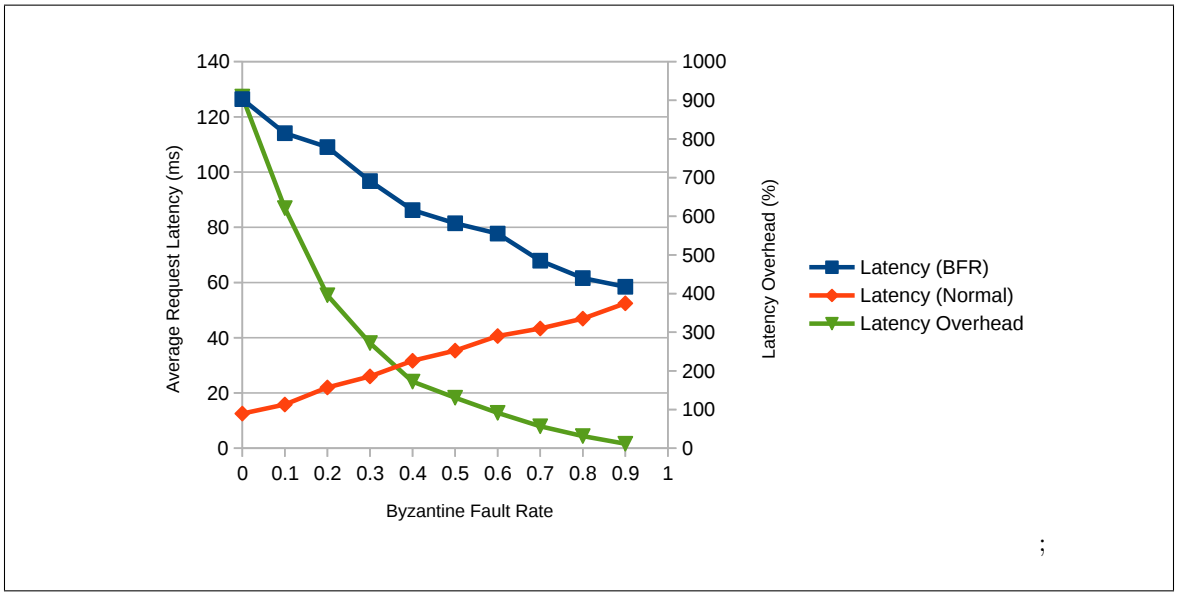


Figure 7.8: The effect of Byzantine failures on latency

7.5.2 Memory

Figure 7.10 shows the effect of Byzantine failures on peak memory usage in the system. The reason that the peak memory usage goes down as the failure rate increases is that, since the system rolls back all faulty updates, the higher percentage of faulty updates means the lower number of updates that remain on the server at any one time, leading to smaller memory footprints for higher failure rates as compared to the non-Byzantine implementation which cannot detect or recover from the Byzantine failures and has to store all previously seen updates. Figure 7.11 shows a similar trend for the effects on peak memory usage for much lower failure rates.

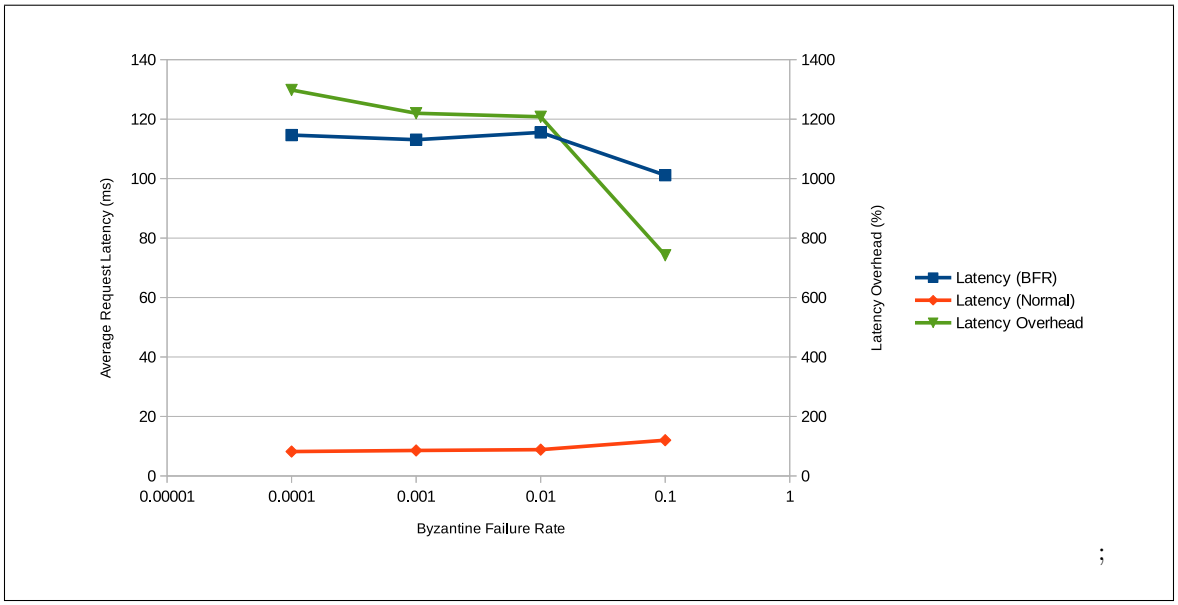


Figure 7.9: The effect of a low rate of Byzantine failures on latency

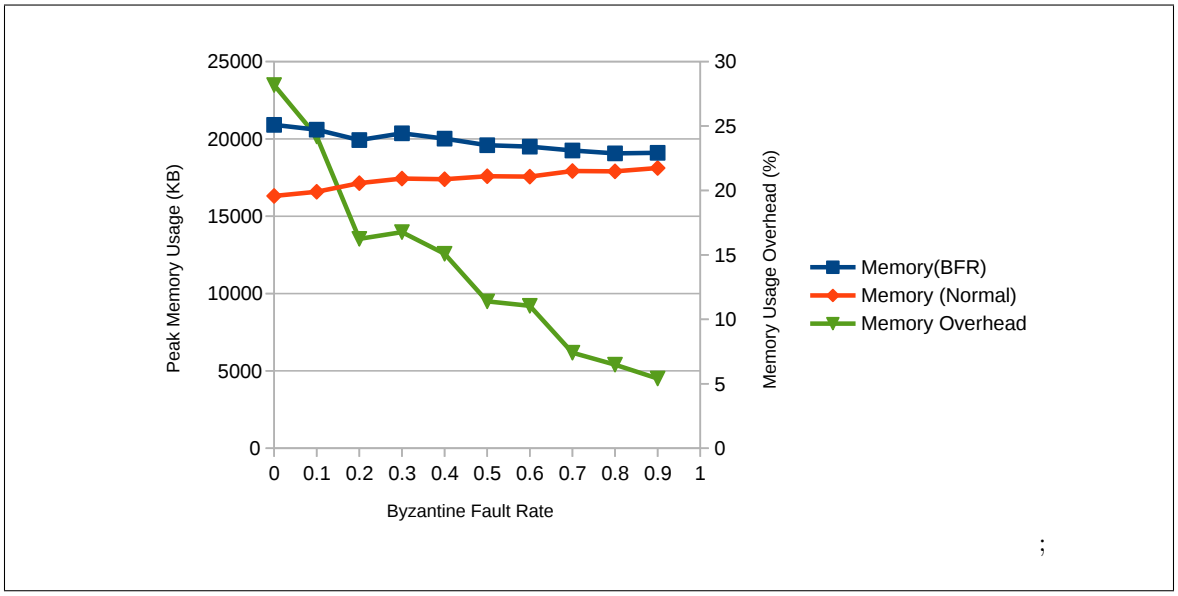


Figure 7.10: The effect of Byzantine failures on peak memory usage

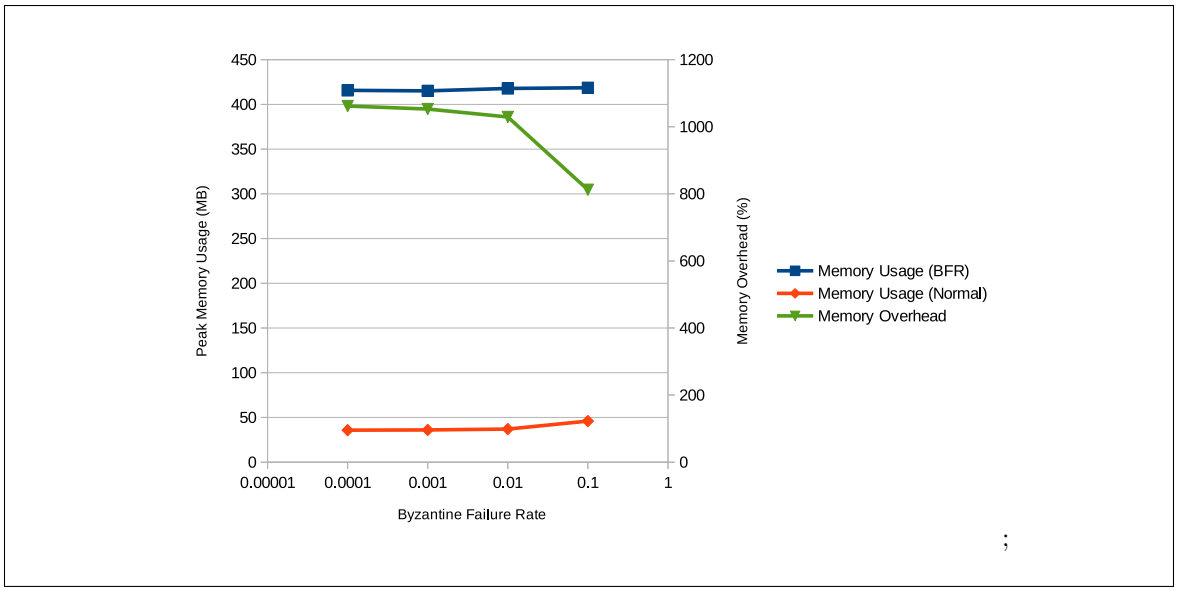


Figure 7.11: The effect of a low rate of Byzantine failures on peak memory usage

7.5.3 Total Network Usage

Figure 7.12 shows the effect of Byzantine failures on total network usage. Since the normal implementation cannot react to Byzantine failures, its network usage is unaffected by their presence, since all of the same messages will be sent regardless of how many failures occur. However, the total network usage of the BFR implementation goes up with the number of Byzantine failures since correct nodes are required to spread around evidence of failures to their neighbors. Even in the extreme case of 90% rate of Byzantine failures, the network usage overhead is still a small linear factor (around 2.8x) compared to the normal implementation. In the case of low failure rates, as shown in Figure 7.13, a similar trend can be seen.

7.5.4 Recovery

Frequency of Rollbacks

Figure 7.14 shows the number of rollbacks as a function of the rate of Byzantine failures in the system. Rollbacks in the system occur as soon as a Byzantine fault is detected. In our implementation, as described in 7.1, each Byzantine failure is immediately detected by each node. As soon as the failure is detected, the system rolls back to the correct state before the fork. As expected, the number of rollbacks increases proportionally to the rate of Byzantine failures since each new Byzantine failure causes a separate rollback in the system. The growth of this quantity could be curtailed in practice by utilizing batching, the optimization mentioned in [2] which allows delaying evidence verification for times of low system load. This delayed

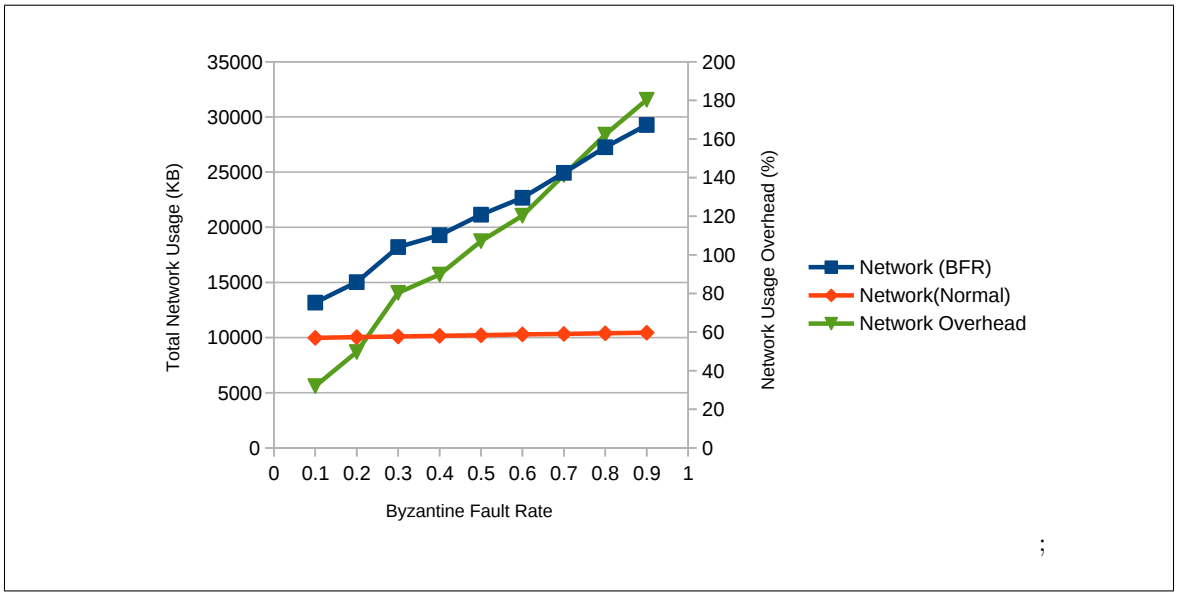


Figure 7.12: The effect of Byzantine failures on total network usage

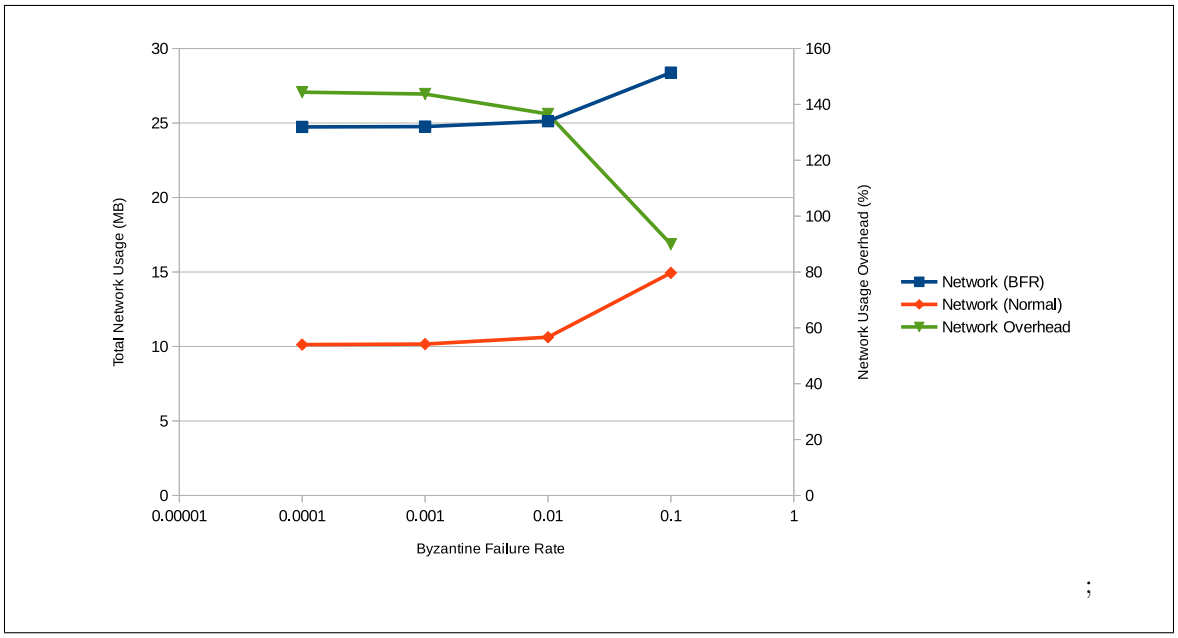


Figure 7.13: The effect of a low rate of Byzantine failures on total network usage

checking would allow a single rollback to eliminate multiple Byzantine failures from the system state. As can be seen in Figure 7.15, in the low failure rate experiment also, the number of rollbacks follows a similar trend.

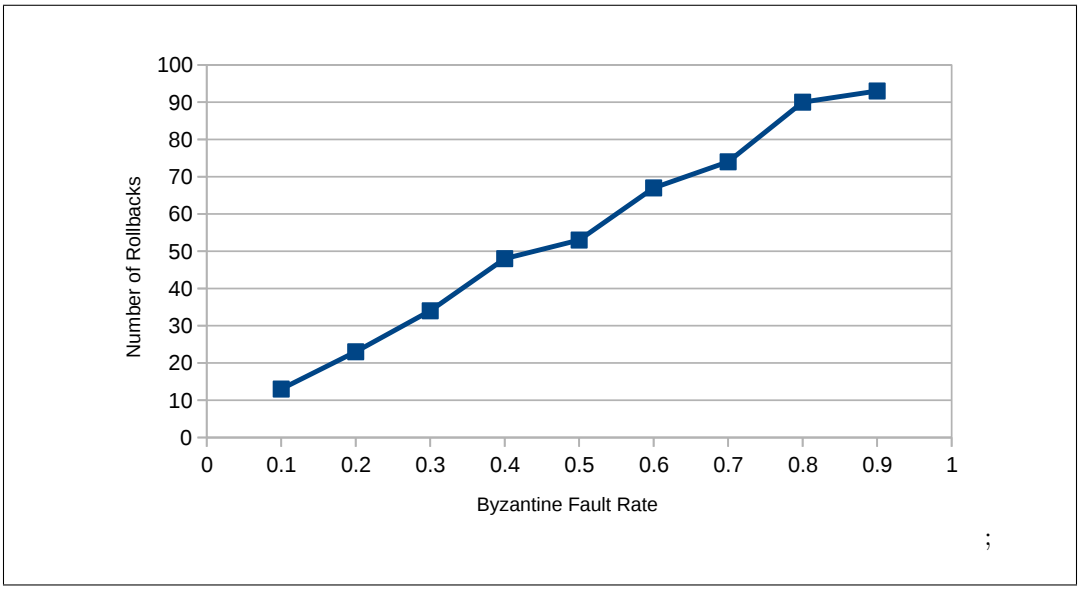


Figure 7.14: Number of rollbacks as a function of the rate of Byzantine failures

Rollback Duration

Figure 7.16 shows the effect of the failure rate on the duration of rollbacks. The average rollback time is inversely related to the failure rate. The reason for this is that when failures are more common, each rollback involves less updates due to the fact that every faulty update is rolled back from the system. This shows that although rollbacks are more common in the case of high failure rates, they also occur with shorter duration which means that the overall time spent performing rollback is relatively constant as the failure rate increases as shown in Figure 7.18. The results of the low failure experiments show a similar trend and can be seen in Figure 7.17.

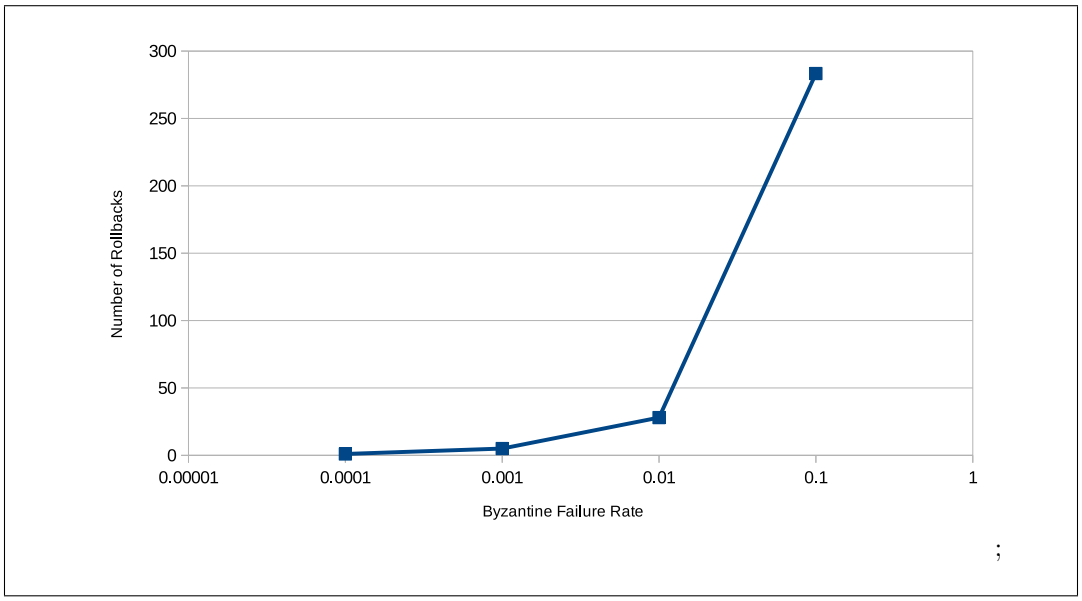


Figure 7.15: Number of rollbacks as a function of the rate of Byzantine failures for low failure rates

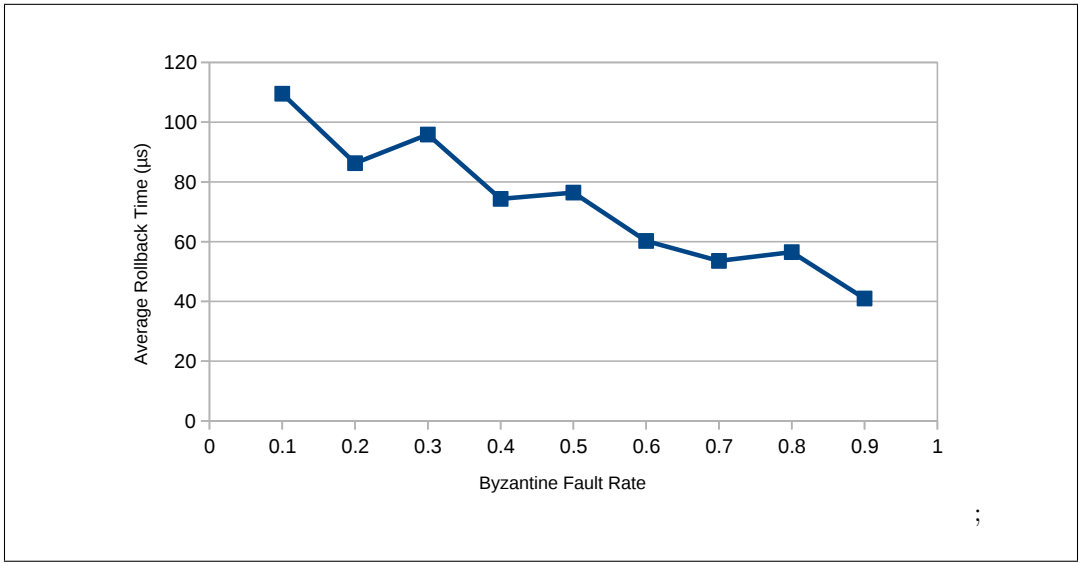


Figure 7.16: Average rollback time as a function of the rate of Byzantine failures

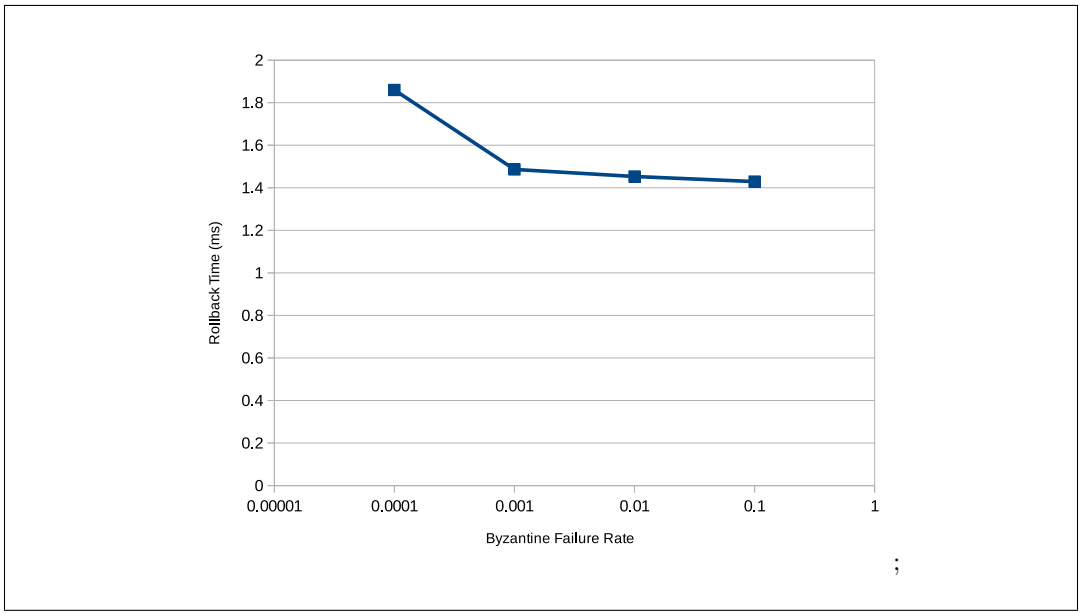


Figure 7.17: Average rollback time as a function of the rate of Byzantine failures for low failure rates

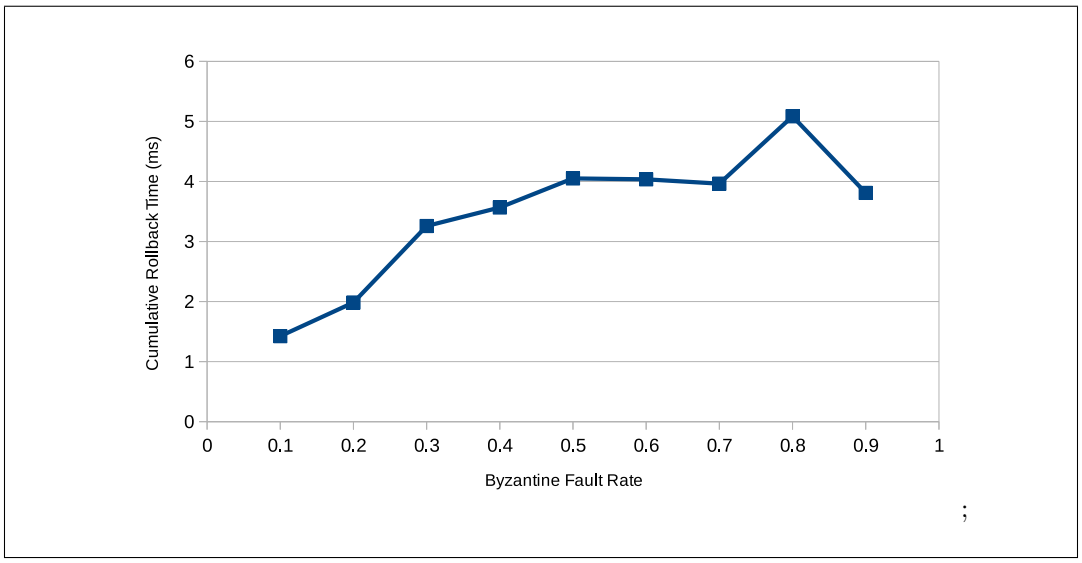


Figure 7.18: Cumulative rollback time as a function of the rate of Byzantine failures

Chapter 8

Discussion

The motivation for this work was to design a system that allowed for applying updates immediately with minimal effect on latency, and then checking for Byzantine faults offline. This was motivated by the idea that Byzantine faults are rare in practice but it might be interesting to provide some eventual guarantees for recovery. Through exploring the related work for Byzantine fault tolerance, many common themes arose, namely the server limitation of Byzantine fault tolerance due to its agreement based protocol. It was not clear how this algorithm could be modified to achieve scalability or responsiveness. Our initial approach was a simple optimistic protocol that committed updates immediately but performed the practical Byzantine fault tolerance protocol [1] in the background. The problem with this approach was that it inherently had the scalability limitations present in Byzantine Fault Tolerance protocol as well as the high network and memory overhead associated with these sorts of agreement protocols.

A major influence on our work was the previous work on the Byzantine Detector [2], which provided many of the guarantees that we were looking for: the ability to eventually detect Byzantine faults with out having to run an online agreement protocol, as well as the nice property of scalability. Since we not only wanted detection of Byzantine faults, but also recovery, this became the focus of our work. This work shows that it is possible to have a system with strong fault tolerance guarantees without sacrificing scalability. The system, however, does not attempt to mask faults as faults can be exposed to the client through lapse consistency. The fault tolerance of this system can be thought of as an eventual guarantee, similar in concept to that of eventual consistency. As these faults are rare in practice, we modeled our system to make sure that performance and consistency is maximized in the common case with no Byzantine faults. If adopted in practice, this system could act as a safety net for Byzantine faults and provide an extra layer of security to applications that currently eschew these guarantees in favor of scalability and performance.

Chapter 9

Future Work and Conclusions

There are many aspects of this paper that deserve a more thorough exploration in future work. One important area would be to consider the interface of the system more in depth. A production ready system requires a configurable interface that allows users to specify their state machine in an imperative or declarative manner instead of having to edit the source code of the application. This would be important for this system to see any adoption in industry.

This thesis covered checkpointing and recovery as a strategy to respond to Byzantine detection, however there are many more avenues for research in this area including providing user alerts, visualizations, policy control, etc.

Our protocol requires total ordering for authenticators to be a valid method of verifying and auditing other nodes. One interesting area of future research would be whether this constraint could be relaxed to weaker orderings such as FIFO or causal ordering which would be more common in eventually consistent systems.

Adaptive optimization would also be a very interesting research topic in this area. Since the batching optimization would allow for nodes to delay evidence checking, this parameter could be automatically adjusted to reflect the current failure rate of the system or to reflect day night cycles and be sensitive to changes in system load.

This work has demonstrated various approaches to recovering from Byzantine failures in a distributed system using Byzantine fault detection. With the threat of cascading failures that can bring down entire clusters due to misconfiguration and software bugs, the need to respond to Byzantine failures is becoming increasingly important. This work provides a solution that does not require the system to choose between scalability and fault tolerance. We have demonstrated that the system provides the ability to react to Byzantine faults when they occur while minimizing the overhead and maximizing the consistency guarantees of the approach during the normal case when Byzantine faults do not occur. We have also shown the scalability of the approach both in Byzantine failure and non failure scenarios.

Bibliography

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002, ISSN: 0734-2071. DOI: 10.1145/571637.571640. [Online]. Available: http://doi.acm.org/10.1145/571637.571640.

[2] A. Haeberlen, P. Kuznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, 2007.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922.

[4] *Hbase*, https://hbase.apache.org/, 2015.

[5] *Riak*, http://basho.com/products/, 2015.

[6] *Voldemort*, http://www.project-voldemort.com/voldemort/, 2015.

[7] *Summary of the amazon ec2 and amazon rds service disruption in the us east region*, http://aws.amazon.com/message/65648/, 2011.

[8] R. Johnson, *More details on today's outage*, https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, 2010.

[9] *The planet blows up*, http://www.availabilitydigest.com/public_articles/0309/planet_explosion.pdf, 2008.

[10] *Help! my data center is down!* http://www.availabilitydigest.com/public_articles/0704/data_center_outages-lessons.pdf, 2012.

[11] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, Seattle, WA, USA: ACM, 2014, 7:1–7:14, ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670986. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670986.

[12] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for byzantine fault detection," in *Proceedings of the Second Conference on Hot Topics in System Dependability*, ser. HotDep'06, Seattle, WA: USENIX Association, 2006, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973416.1973421.

[13] T. Distler and R. Kapitza, "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, Salzburg, Austria: ACM, 2011, pp. 91–106, ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966455. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966455.

[14] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, 12:1–12:38, Dec. 2011, ISSN: 0734-2071. DOI: 10.1145/2063509.2063512. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063512.

[15] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05, Brighton, United Kingdom: ACM, 2005, pp. 59–74, ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095817. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095817.

[16] B. Cohen, *Bep-0003: The bittorrent protocol specification*, http://www.bittorrent.org/beps/bep_0003.html, 2008.

[17] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 16–30, Jan. 2013, ISSN: 0018-9340. DOI: 10.1109/TC.2011.221. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.221.

[18] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA: ACM, 2007, pp. 189–204, ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294280. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294280.

[19] D. Zhuo, Q. Zhang, D. R. K. Ports, A. Krishnamurthy, and T. Anderson, "Machine fault tolerance for reliable datacenter systems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14, Beijing, China: ACM, 2014, 3:1–3:7, ISBN: 978-1-4503-3024-4. DOI: 10.1145/2637166.2637235. [Online]. Available: http://doi.acm.org/10.1145/2637166.2637235.

[20] S. Sen, "New systems and algorithms for scalable fault tolerance," AAI3562341, PhD thesis, Princeton, NJ, USA, 2013, ISBN: 978-1-303-09930-4.

[21] D. Behrens, C. Fetzer, F. P. Junqueira, and M. Serafini, "Towards transparent hardening of distributed systems," in *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, ser. HotDep '13, Farmington, Pennsylvania: ACM, 2013, 4:1–4:6, ISBN: 978-1-4503-2457-1. DOI: 10.1145/2524224.2524230. [Online]. Available: http://doi.acm.org/10.1145/2524224.2524230.

[22] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, 12:1–12:34, Jun. 2011, ISSN: 1094-9224. DOI: 10.1145/1952982.1952994. [Online]. Available: http://doi.acm.org/10.1145/1952982.1952994.

[23] https://www.digitalocean.com/, 2011.