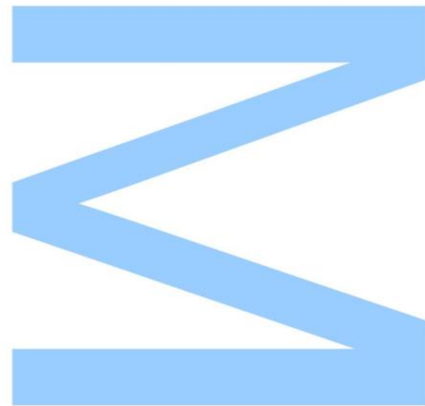
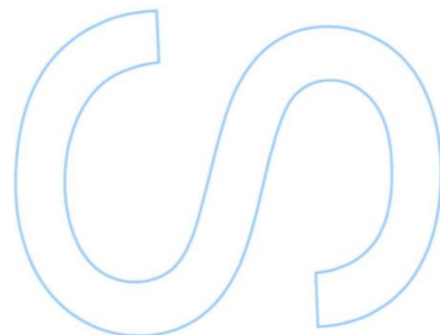
Injecting Faults in Byzantine Fault Tolerant Protocols

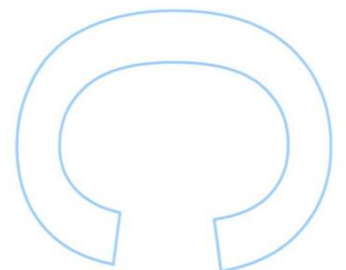
Ricardo Fernandez
Mestrado em Segurança Informática
Departamento de Ciências de Computadores
2021

Orientador
Rolando Martins, Professor Auxiliar, Faculdade de Ciências

Coorientador
João Soares, Professor Auxiliar Convidado, Faculdade de Ciências



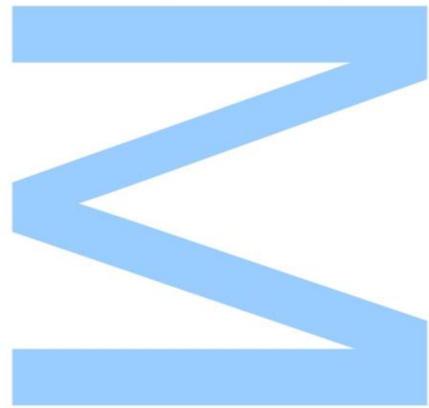
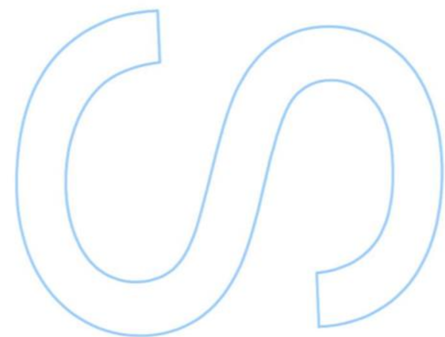
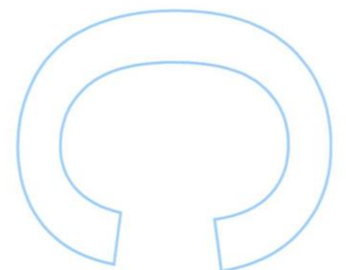




Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Universidade do Porto

Masters Thesis

Injecting Faults in Byzantine Fault Tolerance Protocols

Author:
Ricardo Fernandez

Supervisor:
Rolando Martins

Co-supervisor:
João Soares

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Informatic Security

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências de Computadores

December 7, 2021

“ Do not go where the path may lead, go instead where there is no path and leave a trail. ”

Ralph Waldo Emerson

Acknowledgements

I want to express my gratitude to the following who never stopped helping me until my dissertation was complete :

Professor João Soares, the dissertation Co-advisor, helped me whenever I had doubts about anything related to the subject with his experience in the field and aided me in solving problems in the structure of the thesis.

Professor Rolando Martins, the dissertation Advisor, assisted me on how to construct the base of Zermia and overall work with his field expertise and former experience in the subject.

I also want to thank my family and friends that supported me through this work emotionally.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências de Computadores

MSc. Informatic Security

Injecting Faults in Byzantine Fault Tolerance Protocols

by Ricardo Fernandez

With the ever-growing complexity of computational systems, the ability to sustain faults is a crucial requirement in any system that needs to be available continuously to provide its services. As such, fault tolerance protocols are needed to mitigate the issues caused by faults, being that byzantine faults are the trickier ones to tolerate. Several Byzantine fault tolerance protocols were introduced in the last two decades to deal with this issue in specific environments. However, the development of tools that can test these protocols, notably fault injectors, were rather precarious due to them either not being available to the public or kept under the control of researchers who built those protocols. Moreover, most of the available fault injectors do not work with various BFT protocols, and some of them do not interact or inject failures into clients to explore their impact on these types of systems. As such, the main focus of this document is to introduce Zermia, a new fault injector that uses gRPC as means of communication between server and client, that tries to keep its level of intrusiveness low through the employment of aspect-based functions. The base focus of this work is to facilitate the integration of several protocols with Zermia through gRPC. For the first phase of development of this application, BFT-SMaRt was picked in order to assess what kind of performance it could maintain under the attack of different faults.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências de Computadores

Mestrado em Segurança Informática

Injetando faltas em protocolos tolerantes a faltas Bizantinas

por Ricardo Fernandez

Com a complexidade cada vez maior dos sistemas computacionais, a capacidade de sustentar falhas é um requisito crucial em qualquer sistema que precise estar disponível continuamente para fornecer seus serviços. Como tal, os protocolos de tolerância a falhas são necessários para mitigar os problemas causados por falhas, sendo que as falhas bizantinas são as mais difíceis de tolerar. Vários protocolos de tolerância a falhas bizantinos foram introduzidos nas últimas duas décadas para lidar com esse problema em ambientes específicos. No entanto, o desenvolvimento de ferramentas que possam testar esses protocolos, notadamente os injetores de falhas, foram bastante precários por não estarem disponíveis ao público e ficarem sob o controle dos pesquisadores que construíram esses protocolos. A maioria dos injetores de falha disponíveis não funciona com vários protocolos BFT e alguns deles não interagem ou injetam falhas nos clientes para explorar o impacto dos mesmos neste tipo de sistemas. Como tal, o foco principal deste documento é apresentar o Zermia, um novo injetor de falhas que usa gRPC como meio de comunicação entre o servidor e o cliente, que tenta manter seu nível de intrusão baixo por meio do emprego de funções baseadas em aspectos. O foco básico deste trabalho é facilitar a integração de diversos protocolos com o Zermia por meio do gRPC. Para a primeira fase de desenvolvimento desta aplicação, o BFT-SMaRt foi escolhido para avaliar que tipo de desempenho ele poderia manter sob o ataque de diferentes falhas.

Acronyms

BFT	Byzantine Fault Tolerant
MAC	Message Authentication Code
SMR	State Machine Replication
PBFT	Practical Byzantine Fault Tolerant
RBFT	Redundant Byzantine Fault Tolerant
OBFT	Obfuscated Byzantine Fault Tolerant
FLP	Fischer Lynch Paterson impossibility proof
API	Application Program Interface
POM	Proof-of-Misbehaviour
ACS	Asynchronous Common Subset
ABA	Asynchronous Binary Agreement
BA	Binary Agreement
RBC	Reliable Broadcast
gRPC	Remote Procedure Call
G-PBFT	Geographic-Practical Byzantine Fault Tolerant
AVID	Asynchronous Verifiable Information Dispersal

Contents

Acknowledgements v

Abstract vii

Resumo ix

Glossary xi

Contents xiii

List of Figures xvii

1 Introduction 1
1.1 Motivation 2
1.2 Proposed solution 3
1.3 Contribution 4
1.4 Document Structure 5

2 Related Work and State of the Art 7
2.1 BFT overview 7
2.1.1 Fault models 8
2.1.2 Byzantine fault tolerance concepts 9
2.1.3 Synchrony in BFT 10
2.1.4 fault handling and recovering techniques 12
2.1.5 Byzantine Generals' Problem and the 3f+1 solution 13
2.2 Practical Byzantine Fault Tolerance 17
2.2.1 Protocol overview 17
2.2.2 Fault free execution 17
2.2.3 Checkpoint system 19
2.2.4 Fault handling while having a faulty primary 20
2.2.5 Fault recovery model 21
2.3 Zyzzyva 22
2.3.1 Protocol overview 22
2.3.2 Fault free execution 23
2.3.3 Fault handling when having a faulty primary 24
2.3.4 Fault handling when there is one faulty replicas 26

2.3.5 Fault recovery model . . . 27
2.4 BFT-SMaRt . . . 27
2.4.1 Protocol overview . . . 27
2.4.2 Fault Free execution . . . 27
2.4.3 Fault handling with a faulty primary . . . 29
2.4.4 Fault handling with a faulty client . . . 30
2.4.5 Fault recovery models . . . 30
2.5 Prime . . . 32
2.5.1 Protocol overview . . . 32
2.5.2 Fault free execution . . . 33
2.5.3 Fault handling when having a faulty primary . . . 35
2.5.4 Fault recovery model . . . 37
2.6 RBFT . . . 37
2.6.1 Protocol overview . . . 37
2.6.2 Fault free execution . . . 38
2.6.3 Fault handling when having a faulty primary . . . 40
2.7 Fault injection . . . 40
2.7.1 BFT fault injectors . . . 42
2.7.2 Summary of the BFT fault injectors . . . 43

3 Analysis of protocols under attack 45
3.1 PBFT . . . 45
3.1.1 Protocol fault analysis . . . 45
3.1.2 Network and other faults . . . 49
3.2 Zyzzyva . . . 50
3.2.1 Protocol fault analysis . . . 50
3.2.2 Network and other faults . . . 54
3.3 BFT-SMaRt . . . 55
3.3.1 Protocol fault analysis . . . 55
3.3.2 Network and other faults . . . 57
3.4 Prime . . . 58
3.4.1 Protocol fault analysis . . . 58
3.4.2 Network and other faults . . . 60
3.5 RBFT . . . 60
3.5.1 Protocol fault analysis . . . 61
3.5.2 Network and other faults . . . 62
3.6 Summary . . . 63

4 Zermia: A fault injector for BFT systems 65
4.1 Zermia's architecture . . . 65
4.1.1 Coordinator . . . 66
4.1.2 Agent . . . 66
4.1.3 Faults . . . 66
4.1.4 Fault schedules . . . 66
4.2 Zermia's model and test cycle . . . 67
4.2.1 Initial configurations . . . 67
4.2.2 Booting process . . . 68

4.2.3 Execution process 69
4.3 Zermia's Implementation details 71
4.3.1 Coordinator 71
4.3.2 Agent 74
4.3.3 Predefined faults in Zermia 75
4.3.4 Schedule conflicts 76

5 Experimental Evaluation 79
5.1 Experimental Setup 79
5.2 Baseline results 80
5.3 Communication-based faults 80
5.3.1 Crashing Replicas 80
5.3.2 Delaying Replicas 81
5.3.3 Message dropper 82
5.3.4 Flood attacks 83
5.4 Protocol specific faults 87
5.4.1 Agreement phase - REQUEST 88
5.4.2 Agreement phase - PROPOSE 90
5.4.3 Agreement phase - WRITE 90
5.4.4 Agreement phase - ACCEPT 91
5.4.5 View change phase - STOP 92
5.4.6 View change phase - STOPDATA 93
5.4.7 View change phase - SYNC 94

6 Conclusion and Future Work 97

Bibliography 99

List of Figures

2.1 Scenario in which the siege fails because a conspirator general confuses a loyal general. . . . 13
2.2 Lieutenant 2 is a conspirator. [30] . . . 14
2.3 Commander is a conspirator. [30] . . . 14
2.4 Impostor lieutenant can not compromise plan. . . . 15
2.5 Traitor commander can not compromise plan.[30] . . . 15
2.6 Conspirator can not compromise plan. [30] . . . 16
2.7 Conspirator compromises plan. . . . 16
2.8 PBFT Normal Execution. [9] . . . 18
2.9 PBFT view change. [9] . . . 18
2.10 Zyzzyva best-case scenario execution [2] . . . 23
2.11 Zyzzyva execution with a faulty primary. . . . 25
2.12 Zyzzyva execution with a faulty primary. . . . 25
2.13 Zyzzyva execution when there is a faulty replica. [2] . . . 26
2.14 BFT-SMaRt normal execution. [24] . . . 28
2.15 Execution with a faulty client.[24] . . . 30
2.16 Sequential checkpointing. [33] . . . 31
2.17 State transfer view of the protocol.[33] . . . 31
2.18 Prime normal execution. [14] . . . 33
2.19 Prime execution under Leader suspect and view changing protocols. . . . 35
2.20 RBFT normal execution. [15] . . . 38
2.21 Instance change of RBFT. [15] . . . 39
2.22 Comparison of the characteristics between some of the existent BFT fault injectors. . . . 43

3.1 PBFT fail scenario 1. . . . 48

4.1 Overview of the Zermia Architecture . . . 65
4.2 Example of a fault schedule through command line . . . 71
4.3 Coordinator service interface (gRPC methods) . . . 72
4.4 Agent status requisition service details . . . 72
4.5 Fault schedule requesting service . . . 73
4.6 Update metrics service details . . . 73
4.7 Test results demonstrated in the Coordinator. . . . 73
4.8 Advice function which is called whenever a replica's main function is started in BFT-SMaRt. . . . 74
4.9 General idea of the fault injection advice. . . . 75

5.1 Results from a fault-free experiment 80
5.2 Results for a crash fault schedule 81
5.3 Results from a delay fault schedule 82
5.4 Results for a message dropper schedule 83
5.5 Results from a flood fault schedule that targeted all system 84
5.6 Results from a flood fault schedule that targeted all system 85
5.7 Results from a flood fault schedule that targeted all system 86
5.8 Experimental results for REQUEST message modification 88
5.9 Results for $F = 3$ for when faulty clients send REQUESTS with different operation ids to different replicas. 89
5.10 Results for $F = 3$ for when faulty clients collude with a faulty-primary to delay the system. 90
5.11 Experimental results for PROPOSE message modification. 90
5.12 Experimental results for WRITE message modification. 91
5.13 Experimental results for WRITE message modification. 91
5.14 Experimental results for STOP message modification. 92
5.15 Results for $F = 3$ for when a correct primary gets replaced via a flood of STOP messages. 93
5.16 Experimental results for STOPDATA message modification. 94
5.17 Experimental results for SYNC message modification. 94

Chapter 1

Introduction

With the ever-increasing power and processing in computation, high-performance systems are composed gradually of more components that can be vulnerable to failures. Many infrastructures, such as telecommunications, industrial control, and stock exchanges, have evolved to the point where a simple failure in one of their components can cost exorbitant sums of money and time. As we become more reliant on those systems, these must remain operational and provide their services even in the presence of faults.

The presence of faults can cause systems to stop responding or deviate from expected behavior, with the former typically corresponding to fail stops and the latter to byzantine faults. These two types of faults are pretty distinct, both in terms of how detectable they are and how consistent/predictable they can be. Byzantine faults are strictly inconsistent and unrestricted, meaning they output unpredictably at irregular times with potential collusion capabilities, making them difficult to detect. Fail stops occur when a component stops to function and, as a result, ceases to send and receive messages, making them easier to detect than their counterpart.

In order to mitigate both types of faults, tolerance techniques have been researched and proposed, allowing systems to remain operational in the presence of faults, preventing malfunctioning components from compromising the entire system. These methods are typically employed in distributed systems. After all, they contain a large number of elements that communicate with one another, whereas in non-distributed, the use of these methods is less common because they are less complex. Furthermore, the usage of fault-tolerance in distributed systems allows for a reduction in the likelihood of component and communication failure, which, for example, messages lost during communication could result in a deviation from the system's original purpose. In distributed systems, error propagation

is one of the potential risks [1], considering that a faulty component can spread its issues across non-faulty elements in the system. Another issue that arises is that communication between components needs to be reliable, which means that each correct message must be sent according to the stipulated time-bounds and received accordingly. Lastly, there is another issue to consider which is the hardware. Hardware suffers from being physically related to its wear and time, and as such, it can become defective and potentially flaw the software that runs with it.

Due to the nature of byzantine faults being hard to be detected and eliminated, byzantine fault-tolerant protocols have emerged with the help of state machine replication. Over the years, several protocols have been conceived to solve these sorts of issues, each with its own set of advantages and disadvantages (trade-offs) based on their working environment. The array of protocols is quite diverse, being that the following solutions have been proposed: a) speculation [2–4], b) client reliant [5], c) quorum [4, 6, 7], d) trusted component reliant [8], e) agreement phase focused [9, 10], f) adaptative [11], g)primary focused [12–15] ,and h) randomization [16–19].

Each one of these protocols has its strengths and weaknesses (which we will discuss throughout this document), but certain ones did not explore the entirety of what they can achieve under different types of attacks. As a result, there is a necessity to develop tools that could help both new and existing protocols, as they add a degree of security when being deployed to authentic systems.

1.1 Motivation

Existing BFT protocols attempt to solve a particularity set of problems based on the environment they are set in, whether to improve throughput, latency, robustness, safety, or other purposes. In order to reach such objectives, these protocols focused on characteristics that model them differently than others, such as the employment of primaries, the cryptographic algorithms used (for example, Message Authentication Codes and public-key cryptography), the synchrony model employed (weak, partial synchrony, or asynchrony), the contention level they can accommodate and other properties.

Each solution can be susceptible to specific faults/attacks, either as a result of incorrect implementations(for instance, harmful code) or vulnerabilities in the solutions used(for example, faults/attacks more effective against some solutions than others). Some theoretical models are not straightforward to implement, leading to complex solutions that may

introduce weaknesses or incomplete implementations. One such example is Zyzzyva [2], which had difficulty enforcing the entire protocol in practice due to the second phase of the protocol being very complex, resulting an incomplete implementation.

These reasons suggest that existing BFT protocols were not thoroughly tested in their comfort and discomfort zones. Some of the existing BFT protocols could have achieved better design implementations (due to fewer minor flaws) and mitigated more faults with more extensive and diverse testing. Unfortunately, many of these designs also tend to discredit the client's role in the protocol, making barely any mitigation aspects towards possible malfunctioned clients or malicious ones that could collude with replicas to undermine a system further.

The solution to these predicaments relies on tools capable of testing these protocols in diverse scenarios, such as fault injectors. However, another problem arises: there is a lack of these tools to test BFT protocols in general, specifically ones that give freedom to researchers/users to employ diverse and unique scenarios to identify possible design flaws in their implementations.

1.2 Proposed solution

Fault injection can be used as one of the tools for assessing the reliability, consistency, performance, availability, and security of distributed systems. The injection of faults allows studying the behavior, performance impact, and effectiveness of fault tolerance mechanisms in a system that employs such techniques. BFT Fault injectors are something that is lacking in this specific field, as the existing tools are restricted to use in specific protocols[20, 21] or are not available to the public [22], or they are incapable of injecting faults into clients of a protocol. Having an adaptative tool capable of injecting faults in both clients and replicas on old and possibly new distributed type protocols is something on-demand in this field.

Hence, in this paper, we introduce Zermia, a modular Java-based fault injector designed to allow users to inject faults into BFT systems for testing purposes. It employs a client/server model, where the server acts as a coordinator for the Agents(runtime clients) that run alongside the clients/replicas of a targeted protocol. Additionally, it employs gRPC to reduce communication complexity between the Coordinator and Agents. Fault injection happens through the use of aspects(Aspect-Oriented Programming) based classes in the targeted protocols, which allows us to keep the level of intrusiveness to a minimum.

The approach used in this fault injector allows users to build their fault schedules for each faulty client/replica, where different faults can be applied in the same interval of time. Additionally, when more than one fault is scheduled in the same interval of time, users can enable a priority fault system, in which some faults are run before others to maximize potential system damage. Furthermore, users can also focus the injection of the faults into specific messages and target them against, for instance, the primary, non-primaries, or even the entire system. Through these methods, the need to study multiple BFT protocols is required, as it will provide insight on which faults are more prone to disrupting protocols and add those same faults to this injector.

For the evaluation of how this fault injector affects a BFT protocol, BFT-SMaRt [10] was selected to be tested. Through this work, we check how different types of faults affect the performance of the chosen BFT protocol.

1.3 Contribution

In this document, we make several contributions, being those the following :

- We examine the design of several known BFT protocols and verify their weaknesses in order to identify which faults are more likely to influence the overall performance of BFT protocols.
- We describe the architecture, design, and process of a fault injector that allows users to build their own fault schedule for each faulty client and replica, given that multiple different faults can be applied in the same period of time. The primary capability of this BFT fault injector is the proper injection of multiple faults through the use of aspects (Aspect-Oriented Programming). We can keep intrusiveness low through this approach since there is no need to modify how the targeted protocols interact with the fault injector internally. Furthermore, it employs gRPC[23] as the foundation to exchanging information between server and clients in Byzantine Fault Tolerant protocols, considering that this decreases the communication complexity between them. With the use of gRPC, this fault injector will also be able to work easier with different types of protocols, but that is not the case for now, due to how some of these protocols do not have their code online or because they are buggy.
- We have implemented the fault injector described in this paper into a well-known BFT protocol, BFT-SMaRt [10, 24].

- We exhibit the testing results of the performance impact that network and communication faults can have when injected in BFT-SMaRt.
- We present the testing results for experiments that involved targeting specific messages and functions, such as the agreement and view change phase and the usage of faulty clients to compromise the protocol.
- Paper publication - ”ZERMIA - A Fault Injector framework for testing Byzantine Fault Tolerance protocols” in the NSS2021, 15th International Conference on Network and System Security, held on the 22nd - 24th of October 2021, in Tianjin, China.

1.4 Document Structure

This dissertation is divided into six chapters, one of which we are at the moment. In Chapter 2, we examine how various BFT protocols work. Chapter 3 discusses some attacks that, in theory, should be possible (and have been proven) against the protocols discussed in Chapter 2. In chapter 4, we present Zermia, a fault injector, and how it is structured. In chapter 5, we investigate which faults were more effective in disrupting BFT-SMaRt through Zermia. Lastly, in chapter 6, we summarise the work done in this document and possible improvements in the future.

Chapter 2

Related Work and State of the Art

In this chapter, we will discuss the principles behind Byzantine fault-tolerant protocols. We start by introducing the fundamental theoretical concepts of BFT systems. Following that, we investigate the different approaches taken by various BFT protocols in order to determine what characteristics distinguish them from one another. The explanation of each protocol will also provide insight into what types of faults have an effect on the performance of these systems.

2.1 BFT overview

In the context of distributed systems, replication has been used to improve service availability. This technique improves the system's resilience by allowing it to forward operations to active replicas in the event that one is absent or fails. Through this method, we get more performance, availability, and the ability to tolerate faults. Maintaining multiple active or backup copies of a service's logic and data allows systems to cope with faults by replacing crash copies with active ones.

There are two principal types of replication in distributed systems, active and passive. In active replication, replicas execute all operations in the same order. One example that falls into this category is state machine replication [25]. In SMR, each replica is handled as a finite state machine. Each operation executed by a replica changes its state from an initial one to a final one, where both have to be valid. Since replicas execute all operations in the same order, it requires them to be deterministic, meaning that replicas state must update to the same predetermined state with each operation execution, given that any unexpected deviation would imply that a replica is malfunctioning.

In the case of passive replication, also known as primary-backup, only one of the replicas performs all operations, and this replica is referred to as the system's primary. Whenever a client sends a request to the system, the primary is the one who receives and processes it, sending the result back to the client. Additionally, the primary also sends the client's result to non-primaries replicas (backups) to update their state. If the primary crashes, one of the non-primaries replaces it, takes control, and becomes the new primary. One of the issues about this type of replication is that faulty primaries and non-primaries can compromise the system's state by not behaving according to their original specifications. To mitigate these issues, all replicas in the system must execute the same operations requested by the clients and send the same results to them.

2.1.1 Fault models

Faults are the most common cause of systems deviating from their normal behavior, as they can cause, for example, a replica to stop working. There are several types of faults that can cause such deviations, including the following:

- Crash faults: They manifest whenever a component in a system fails to function. This behavior is detectable since it is unusual for a component to stop sending messages or receiving them. In other cases, there are support systems that can detect such problems and send alerts when a component fails. Crash faults are classified into two types: fail-stop and fail-recover. Where in the former, failed components never automatically recover but can be manually restarted, and as for the latter, crashed components are recoverable through an automatic mechanism after a set period of time.

- Timing faults: They occur whenever a component in a system fails to produce results within an established time frame. That is, if a component sends a message earlier or later than the established time interval, the system may ignore potentially vital messages or deviate from its normal behavior.

- Omission faults: These occur whenever a component fails to produce results that were intended to be executed. Examples of this fault include a component attempting to send or receive a message but failing to do so because the messages never made it to their respective I/O buffers, which could be due to a buffer overflow.

- Responsiveness faults: These happen whenever a component fails to produce expected results. Examples of such are when a faulty component sends a message with the wrong values attached to it.

- Arbitrary/Byzantine faults: These faults are highly unpredictable and inconsistent, which means that anything can happen at any time, as malicious actors who exploit vulnerabilities to impact a component or a system are also included in this fault model. This inconsistency is due to Byzantine faults encompassing every single other fault in its class, making it the most widespread fault model of them all. In addition, byzantine faults can also cause collusions between faulty components, further destabilizing a system and making mitigation more challenging to accomplish.

Another consideration with faults is that their persistence in components can make them more difficult or easier to detect. There are three fault persistency classifications, which are as follows:

- Transient faults: These faults trigger once and then vanish after.

- Intermittent faults: This type of fault appears and disappears at either random or timed intervals of time. Because of such, they are the most challenging faults to detect and identify.

- Permanent faults: These faults regularly activate until the process of recovery/mitigation occurs.

2.1.2 Byzantine fault tolerance concepts

Byzantine faults tend to be the most problematic of these fault types for being capable of anything and everything, making them troublesome to eradicate. As a result, dedicated fault-tolerant techniques have been proposed and used in conjunction with SMR to mitigate byzantine faults, where they aim to reach consensus and secure availability.

BFT-SMR protocols ensure two properties under faulty and fault-free environments, safety and liveness. According to Cachin et al. [26], these two properties satisfy and accomplish the following :

- Safety, which ensures that replicas that execute the same set of operations in the same order reach the same final state and produce the same result.

- Liveness, which guarantees that all requests from non-faulty clients are eventually executed and decided on a value by the consensus group.

As explained by Cachin et al. [26], a correct consensus protocol, when achieved, must guarantee the following four properties :

- Termination: Every non-faulty replica eventually learns about some decided value.
- Agreement: All non-faulty replicas must agree on the same value.
- Validity: If non-faulty replicas propose a value, then all non-faulty replicas eventually decide on the same value.
- Integrity: Each non-faulty replica can only decide on one value. If it decides on a value, then all other non-faulty replicas must have proposed the same value.

The properties here described are achievable based on the concepts of safety and liveness, where the first property is attainable in the presence of liveness and the remaining in the presence of safety. However, these attributes can be violated when deterministic replicas are unable to accurately participate in the consensus group due to, for example, executions that are out of place/time.

2.1.3 Synchrony in BFT

BFT protocols differentiate themselves into two major groups, deterministic and non-deterministic. The existence of these two classes is due to the FLP impossibility [27] result, which states that consensus is not achievable deterministically in an asynchronous environment if a single process can crash, meaning that any deterministic algorithm cannot reach a decision when combined with asynchrony and the possibility of a failure. Another important aspect of this proof is that it is impossible to distinguish between a crashed replica and a slow-moving one in an asynchronous setting.

To overcome the problem posed by the FLP impossibility, several techniques were developed and combined in order to achieve consensus. One of these methods proposed that protocols adopt different communication models, such as the following:

- Synchronous models are the inverse of asynchronous models, being that there are time assumptions regarding communication and processing. As such, in a BFT system, each replica must to send and receive messages within a known and fixed time-bound.

- Weak synchronous models have an upper time-bound that varies. It increases the time-bound in order to satisfy liveness. The predicament with this is that if the time-bound is large enough, faulty primaries can abuse this flaw.

- Partial synchronous models are systems that, depending on the situation, are both synchronous and asynchronous. It can behave asynchronously for periods of time, primarily due to message delaying/processing, but will try to stabilize and act synchronously. The upper time-bound in these models is unknown, but it must be satisfied in some way.

- Random Synchrony models are systems that deliver messages with random time delays. This delay has an associated upper bound in terms of time. The dilemma with this model is determining whether a replica has crashed or is delaying the transmission of a message.

- Asynchronous models are systems that do not make assumptions about timing properties, this type of system does not establish an upper bound for message processing and transmission delays. When delivering messages in a BFT system, each replica has no concept of time, but they are still required to send messages to other replicas. Models like these are effective against malicious actors attempting to stall communications.

According to Correia et al. [28], other strategies used to bypass FLP involved sacrificing determinism (randomization) and the use of oracles (failure detectors).

Through these techniques, non-deterministic protocols were introduced in practice. These protocols typically solve consensus by combining a probabilistic and asynchrony approach with a dependable broadcast protocol (for example, Bracha's broadcast protocol [29]) for communication. In this type of system, the probability that a correct replica is undecided after k rounds approaches 0 as k approaches infinity.

On the other hand, deterministic protocols rely on reaching consensus at each round before moving on to the next. This allows them to establish a point of reference before proceeding and ensure safety. As for synchrony models, this type of protocol primarily employs partial and weak synchrony.

2.1.4 fault handling and recovering techniques

Fault tolerance techniques are classified as dynamic or static redundancy. In dynamic redundancy, the system can be reconfigured to respond to new faults. New replicas, for example, can join a consensus group to replace a faulty replica without reconfiguring the entire system. As for static redundancy, the system employs techniques that can correct faults or isolate faulty replicas but cannot be reconfigured without restarting the entire system with the new configurations.

Most BFT protocols, predominantly deterministic, use a primary that receives requests and distributes them with the non-primaries. The major problem is that the primary can be faulty. When such situations happen, the primary must be replaced, so to prevent a faulty primary from compromising a system's liveness, view change protocols are used. These protocols' main goal is to elect a new primary to resume request processing. The standard process relies on changing the view to a new one. A view is a set of configurations for the system to work with, consisting of the primary and non-primaries. A single view is used in several instances of the protocol until a faulty primary is noticed. When that happens, a new view is created with a new set of primaries and non-primaries. The election of a new primary is usually through a round-robin scheme [2, 9, 13]. In some protocols [13, 14], the former faulty primary can enter into a blacklist and be unable to become the primary for a given time. Non-deterministic protocols do not require such replacement protocols because since they are asynchronous, they do not rely on timers and progress whenever messages are delivered. The former does not apply to deterministic protocols since once a primary fails to deliver a message in a specific time interval, it gets replaced via view change.

Recovery protocols are the mechanisms used by replicas to restore their own state to a fault-free state. These protocols are made of use when a replica is added to the system(which it will need to update its state to be on par with the other replicas) or for recovering a faulty replica. Some tools involve the usage of logs of the actions performed by replicas. These are created at each k-interval and called checkpoints once verified by other replicas. A replica that needs to update its own state can then query other replicas for the most recent stable checkpoint and build its own log from there. Through this, we are bringing this replica to a state prior to the occurrence of the fault. State transfers are another tool used, in which replicas transfer their states to a recovering replica in order to bring its state up to date.

In order to minimize the occurrence of faults, a proactive replica recovery protocol[9] was proposed, which periodically reboots replicas and installs a clean image every k-period of time.

2.1.5 Byzantine Generals' Problem and the 3f+1 solution

The Byzantine general's problem[30] is a fictional plot in which a dispersed Byzantine army attempts to engage an enemy city. This scenario attempts to answer how several distributed entities can reach an agreement before taking action, as well as how many correct entities are required for each malicious/faulty actor in order for an agreement to be still reachable.

The Byzantine army is divided into divisions, each led by a different general. The generals use a messenger to communicate with one another in order to reach an agreement on a strategy (attack or retreat). The issue here is the possibility of traitors among the generals. The conspirators may mislead the loyal generals and cause the joint strategy to fail, resulting in some generals attacking the city while others retreat, resulting in a siege failure against the enemy city.

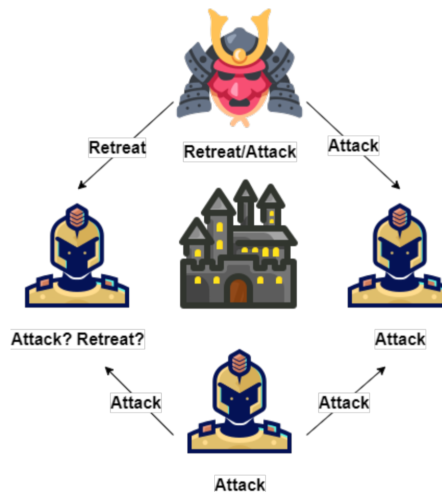


Figure 2.1: Scenario in which the siege fails because a conspirator general confuses a loyal general.

As illustrated in Figure 2.1, if a conspirator decides to send a different order to a different loyal general, the order will be conflicted, resulting in a failed siege because there was no complete agreement among all generals.

The solution to this problem entails using a single commanding general who issues orders to all of his $n-1$ lieutenants, provided that the following two interactive consistency [30] conditions are met:

- IC1. All loyal lieutenants obey the same order.
- IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Based on the conditions stated above, we can conclude that condition 1 is verified by condition 2 if the commanding general is loyal. The solution used replicates the general's orders in each lieutenant, which serves as the basis for reaching an agreement.

As stated by Lamport et al. [30], this problem cannot be solved with only three generals in total. The following figures 2.2 and 2.3 demonstrate why it is not viable:

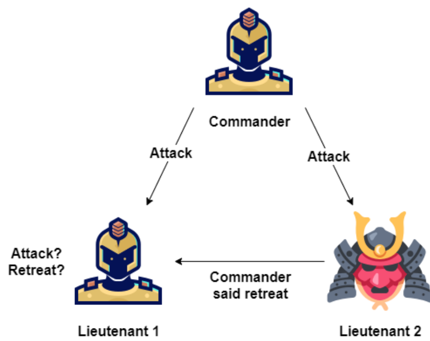


Figure 2.2: Lieutenant 2 is a conspirator. [30]

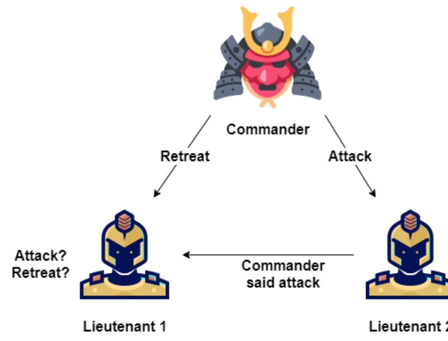


Figure 2.3: Commander is a conspirator. [30]

As shown in the illustrations 2.2 and 2.3, it is impossible to have a clear case of consensus when one of the generals is a conspirator because it will consistently deliver the wrong order to one of the loyal lieutenants, causing conflict. Through these examples, Lamport et al. [30] state that no solution with less than $3f+1$ generals can withstand f conspirators, but it is possible to circumvent with the addition of signatures.

One solution to this problem, according to Lamport et al. [30], is to use $3f+1$ or more generals, denoted as "solution with oral messages." In order for this solution to work, the following assumptions must be made:

- A1. Every message that is sent is delivered correctly.
- A2. The receiver of a message knows who sent it.
- A3. The absence of a message can be detected.

These assumptions prevent conspirators from interfering with message transmission (A1 and A2) and from not communicating with the other parties (A3). The following figures 2.4 and 2.5 show instances where the impostor is unable to compromise the consensus, where "X", "Y" and "Z" represent different messages. It is worth noting that lieutenants save every message they receive in a set M, and if a type of message appears the most, it is used as an order; otherwise, all lieutenants retreat as there was no agreement.

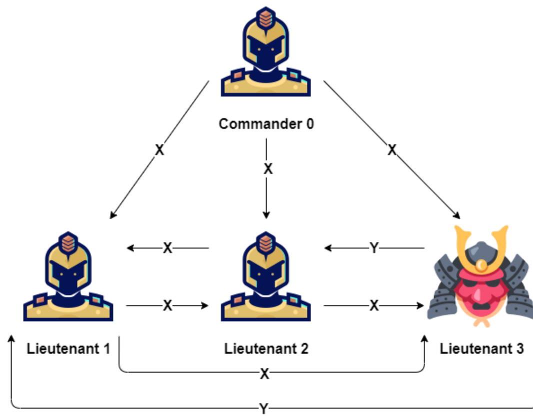


Figure 2.4: Impostor lieutenant can not compromise plan.

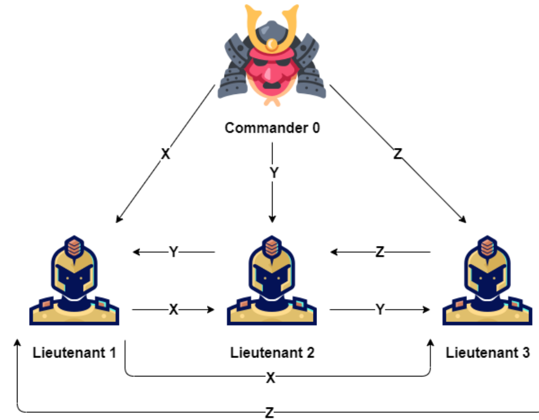


Figure 2.5: Traitor commander can not compromise plan.[30]

The illustration in figure 2.4 shows how the lieutenant conspirator cannot deceive any of the lieutenants with a misleading order. In this example, after the loyal commander sends the "X" order to all lieutenants, the lieutenants consequently exchange messages between themselves and try to reach and agreement. The issue here is that there is a conspirator lieutenant who tries to mislead the other lieutenants by sending "Y" orders that differ from those issued by the commander. This attempt by the conspirator fails because both lieutenants 1 and 2 received $M = (X, X, Y)$ orders, where the "X" order was agreed to be executed as it was in the majority. It is important to note that both interactive consistency IC1 and IC2 are respected, since all loyal lieutenants obey the same order and the order issued by the loyal commander.

A conspirator commander can also try to mislead loyal lieutenants in following wrong orders, but this attempt also fails as shown in figure 2.5. In this example, the conspirator commander relays different orders to all lieutenants, which then they exchange messages between themselves. The lieutenants perceive that the commander is malicious as they received $M = (X, Y, Z)$ different orders after exchanging messages between themselves, resulting in no agreement because all orders had the same magnitude of the majority value.

The other solution to this problem revolves around the use of signatures to ensure non-repudiation in behalf of all lieutenants and commander involved. According to Lamport et al. [30], this solution uses less generals than the first solution and employs the previous three assumptions A1, A2 and A3, as well as two new assumptions:

- A4. A loyal commander's or lieutenants signature cannot be forged, and any alteration of the contents of their signed messages can be detected.
- A5. Anyone can verify the authenticity of a commander's or lieutenant's signature.

For this solution to work, it is assumed that if each lieutenant's set M (messages) contains only one element, that element will be used as an order; otherwise, if that M set is empty, the lieutenants will consequently retreat.

The following figure 2.6 shows how adding signatures can prevent a conspirator from compromising the operation because the loyal lieutenants verify for order inconsistencies in the messages passed between them. In this case we have a malicious commander that sends two messages signed with its own identifier "0" to each lieutenant. The lieutenants then exchange messages by signing them with their own identifier ("1" or "2") on top of the commander's signature, which they received from the malicious commander. Finally, the lieutenants conclude that the commander is malicious because he signed two different types of messages that could not be forged, resulting in the lieutenants M sets containing $M = (Retreat, Attack)$ orders.

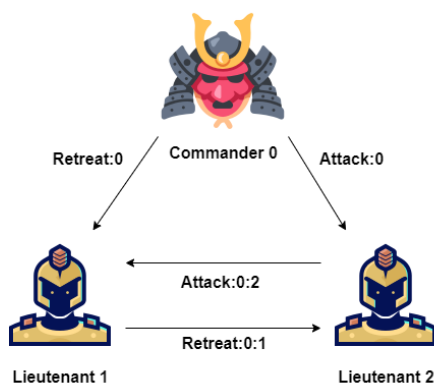


Figure 2.6: Conspirator can not compromise plan. [30]

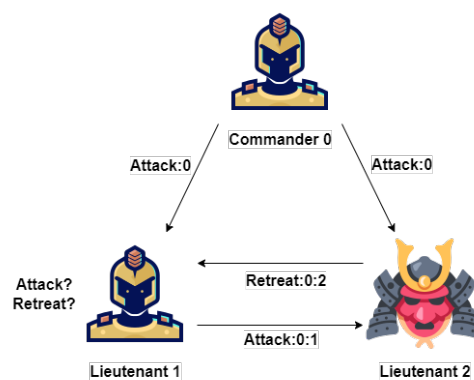


Figure 2.7: Conspirator compromises plan.

Knowing how to do the commander's signature is the unique way for conspirators to jeopardize the plan. Figure 2.7 illustrates how, on the very rare occasion that the conspirator knows how to redo the commander's signature, it can then send contradictory messages

to the loyal lieutenant. In this case, the conspirator lieutenant's "Retreat:0:2" message is a contradictory message signed with the conspirator's signature on top of the commander's forged signature. In this case, the lieutenant 1 M set contains $M = (Attack, Retreat)$ and may suspect the commander of malice.

Lamport et al. [30] claims that these solutions are only applicable in synchronous systems that have a time-bound interval to generate and transmit a message. A malicious actor can compromise a system by delaying the processing of a replica or delaying communication between other replicas until the time-bound is exceeded, putting the system's safety at risk. As a result, these systems must be somewhat synchronous.

2.2 Practical Byzantine Fault Tolerance

2.2.1 Protocol overview

PBFT [9][31] is a primary-based SMR Byzantine Fault Tolerant protocol that has a requirement of $3f + 1$ replicas for guaranteeing liveness and safety in the presence of up to f faults. It uses partial synchrony as the communication model. It employs either MACs or digital signatures for message authentication, provided that authentication happens in every protocol step. Clients are not excluded from the authentication process since they share a secret key with each replica to authenticate client's requests and vice versa.

This protocol was among the first practical BFT solutions designed to work in asynchronous environments(for instance, the Internet). The solution itself was later used as a baseline to several other protocol implementations, such as [2, 7, 10, 12, 14, 32] due to how well it performed. It uses a primary to order requests from clients so that all replicas(primary and non-primaries) execute all operations deterministically.

2.2.2 Fault free execution

According to Figure 2.8, when using MACs as a form of authentication, the normal execution of this protocol consists of the following steps:

1. A client sends a REQUEST message to all replicas, including the operation, a timestamp for once-time semantics, and the client's identifier. Meanwhile, it starts a retransmission timer. This retransmission timer is used if the client does not receive responses before the timer expires. When this happens, the client re-sends the same

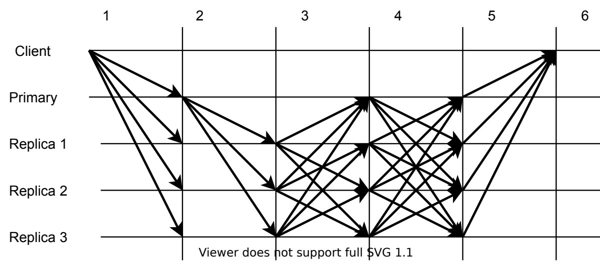


Figure 2.8: PBFT Normal Execution. [9]

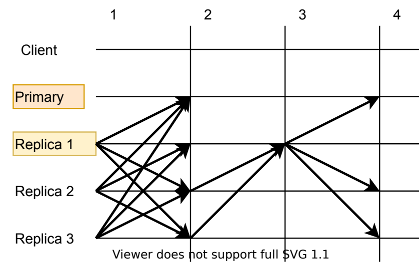


Figure 2.9: PBFT view change. [9]

request to all replicas. Replicas that receive the request, authenticate, save it in the log, start a timer and send it to the primary.

2. When a primary receives a REQUEST message, it authenticates it. If valid, a PRE-PREPARE message is sent to non-primaries, and the primary adds this information to its log. The message includes the view and sequence number(assigned by the primary) and a digest of the client's message.

3. When replicas receive a PRE-PREPARE message, they check to see if the corresponding MAC, view, and sequence number are valid. Following this, a PREPARE message is sent to the other replicas in the consensus group. The message contains the primary's view and sequence number, as well as a digest of the client's request. Both messages are also stored in the non-primaries logs.

4. In this phase, replicas wait for PREPARE messages. When receiving them, they authenticate and review their contents. Once replicas accept at least $2f+1$ matching PREPARE messages, they multicast a COMMIT message. This type of message includes the same view and sequence number as in the PRE-PREPARE and PREPARE message, providing also that they are keen on executing this request since they have received at least $2f+1$ matching PREPARE messages. Replicas save the COMMIT message information into their log.

5. Once replicas start receiving COMMIT messages, they validate the associated MACs and their contents. If replicas receive $2f+1$ matching COMMITs, they can then send a REPLY message to the client. A REPLY message is made up of a view number, the client's request timestamp, the replica's and client's identifier, and the operation's result. Replicas store this information in their log in case they have to retransmit this REPLY to the same client.

6. The client waits until it receives at least $f + 1$ matching REPLY messages from different replicas, in which their MAC is valid, and the contents, more specifically the result and timestamp, match. If the client does not receive a reply within its retransmission timer, it will retransmit its request until it receives enough replies. In case the request had been already executed, and the client keeps retransmitting the request, the replicas retransmit the corresponding REPLY to the client.

2.2.3 Checkpoint system

To prevent replicas from storing an infinitely growing log, PBFT employs a checkpoint system. As such, they discard information on previously executed requests. However, since there is a chance that a replica might require some information related to it (for example, a client requiring a retransmission reply), replicas only discard when they receive a proof that the state on which those requests are based is correct.

Replicas generate checkpoints at fixed intervals of k-requests (for instance, at every 1000 requests). Checkpoints that are created require a certificate proving that they are stable from other replicas. The following list summarizes how to retrieve or create a checkpoint, as previously explained:

1. To retrieve or create a checkpoint, a replica must multicast a CHECKPOINT message to all replicas, which contains the last sequence number used for a request's execution in its state as well as a digest of the state.

2. Replicas wait till they receive $2f + 1$ matching CHECKPOINT messages. Once replicas have collected $2f + 1$ CHECKPOINT messages, they authenticate and verify if the contents have the same sequence number and digest. If everything matches, this checkpoint becomes a certified stable checkpoint.

3. Replicas can then discard former checkpoints and information with a lower sequence number than the one received since the current checkpoint is stable.

During the execution of the protocol, each message that replicas save on the log must contain a sequence number that falls between a specific interval. This number goes from the sequence number of the last stable checkpoint to the sum of it with the log's size (constant number), limiting the information in each replica's log limited. Replicas preserve copies of its current state and last stable checkpoint to prove that it has a stable checkpoint.

2.2.4 Fault handling while having a faulty primary

For the purpose of preventing a faulty primary from compromising the system's liveness, PBFT defines a view change protocol in which replicas agree to a new primary. Non-primaries initiate a view change whenever they accept a request from a client, but they do not receive an ordered request for execution from the primary of the said request within an interval of time. View changes can also start in case a faulty primary sends requests that non-primaries were not expecting to execute. The list that follows goes step by step on how a view change is executed with the assistance Figure 2.9 as well.

1. When non-primaries wait too long to execute a request that they initially received from the client but did not receive a PRE-PREPARE message for that request from the primary, they trigger a view change (Each replica has a timer associated with each request it gets from a client, if it expires, replicas begin a view change). In these situations, non-primaries discard any messages associated with the current view and proceed to multicast VIEW-CHANGE messages. This message holds the next view number (view+1), the sequence number of the last stable checkpoint(h), a set of $2f + 1$ signed CHECKPOINT messages that validate the sequence number and the digest of the checkpoint (both of which belong to C), and two sets of information's related to the requests that had been prepared(P) and pre-prepared(Q) in former views. This phase is one of the few that requires the use of a public key to sign the message.

2. Upon receiving VIEW-CHANGE messages, replicas verify the signature and if the contents, specifically the two sets of information, are from view numbers less than or equal to the actual view. After validating the contents, replicas send VIEW-CHANGE-ACK messages for each VIEW-CHANGE message accepted to the new primary. This message contains the next view, the replica identifier that sent the message, the replica identifier that sent the VIEW-CHANGE message, and a digest of the VIEW-CHANGE message.

3. The new primary agrees on a VIEW-CHANGE from a replica only if it receives $2f - 1$ VIEW-CHANGE-ACK for this replica from other replicas. The new primary uses the gathered information to choose a checkpoint that is going to be selected for request processing on the new view and selects requests to be pre-prepared for each sequence number (between an interval, ranging from the sequence number of the last

stable checkpoint to the sum of it, plus the log size) in the new view. If there is a formerly prepared request by $2f + 1$ replicas that did not get fully committed or did get committed in the previous view, the new primary chooses this request to be ordered. If some requests were not prepared before with a specific sequence number, the new primary assigns it a ”null” to fill gaps. After all sequence numbers have been filled, the new primary multicasts a NEW-VIEW message with its selection. This message consists of a set of identifiers of the replicas who sent the VIEW-CHANGE message plus the digest of their VIEW-CHANGE messages and the checkpoint and the sequence numbers for the requests. It is also signed with a digital signature.

4. With the NEW-VIEW message, replicas can verify if there is any incorrect assignment, and if so, a new View Change for view+2 is initiated. Otherwise, because the NEW-VIEW phase performs a similar function to the PRE-PREPARE phase, the replicas can proceed to the PREPARE phase.

2.2.5 Fault recovery model

In terms of recovery, PBFT has several mechanisms in place to protect the system from malicious actors and recover replicas back to their current state. One of these mechanisms is the employment of a cryptographic co-processor that stores private keys and encrypt and decrypt messages. It also utilizes a ROM (Read-Only-Memory) to save public keys from other replicas safely. Additionally, the ROM stores a copy of the service code and Operative System used to restore when rebooting. MAC keys are changed during recoveries using NEW-KEY messages frequently, and if a client does not change its key, replicas discard their current keys and force the client to refresh its key. Replicas and clients discard any message that uses old keys.

This recovery protocol employs a ”watchdog” timer, which causes replicas to proactively recover on a regular basis when the timer expires. This design principle is critical because a replica may appear to be correct when it is not. Also, for this watchdog mechanism to work correctly, a malicious actor cannot have physical access to it since it could change the replica’s timer to keep making recoveries above a certain threshold which other replicas would not notice.

Recovery begins at a replica when its ”watchdog” timer expires, and it passes the control to a recovery monitor, which stores the replica’s checkpoints, state, and logs to a disk. It then proceeds to reboot the system with a clean operating system and restore

its state, both of which are stored in a Read-only-memory (so it cannot be changed by a malicious actor). Every time a recovery is made, the recovery monitor verifies a digest associated with the correct operative system and state. If it does not match, it requests other replicas for a copy of the correct code in order to perform the recovery. After this process, the replica has to preserve its state and execute requests while making a recovery if necessary. The replica also replaces the MAC keys with new ones with the other replicas.

After this initial step, the recovering replica must run an estimation protocol in order to assess an upper bound (for the log) through the help of the other replicas (checkpoints and log information that have a higher sequence number than the estimated upper bound are promptly deleted). The recovering replica multicasts a QUERY-STABLE message, to which the other replicas respond with a REPLY-STABLE message containing the most recently prepared request and the sequence numbers from the last stable checkpoint. With this information, the recovering replica can establish an upper bound and a minimum bound and proceed to participate in the protocol, but will not be part of any execution that is above the upper bound established until it settles with a new stable checkpoint with a sequence number that is equal or higher to the upper bound.

Finally, the replica's cryptographic co-processor builds a RECOVERY request message that is multicasted. Replicas only receive the message if it is not a replay or if that replica has not recovered in a long time. This message is treated similarly to a client's request, in which it is assigned a sequence number, and as such, it goes through the same three phases of its regular operation. MAC keys are refreshed via a NEW-KEY message when replicas execute the recovery request. Lastly, the recovering replica waits for $2f+1$ replies before calculating its recovery point and view.

2.3 Zyzzyva

2.3.1 Protocol overview

Zyzzyva[2] is a primary-based Byzantine Fault Tolerant protocol that has a requirement of $3f+1$ replicas in order to secure liveness and safety in the presence of up to f faults. Zyzzyva's communication model is partial synchrony. It uses MACs or digital signatures for message authentication, provided that authentication occurs at every step of the protocol, with two exceptions. The first occurs when a client re-transmits a message, and the second occurs during a view change, considering digital signatures are required at those times.

Zyzzyva improves on PBFT[9, 31] by providing a speculative execution model. Under the speculative execution, replicas are assumed without faults, and since replicas do not establish a consensus between them, they must rely upon clients to certify whether the replies and history received from the quorum are stable or not. In the event that they are not stable, the client waits until there is a convergence. If the reply and history are stable, the client proceeds to accept the reply provided. When clients or replicas detect that the primary is faulty, they can initiate a view change, which results in the primary being replaced.

2.3.2 Fault free execution

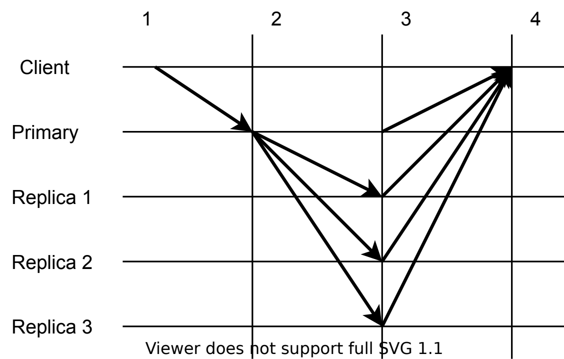


Figure 2.10: Zyzzyva best-case scenario execution [2]

This protocol's standard execution consists of only two steps: an ordering phase performed by the primary and a speculative phase performed by all replicas. Messages are authenticated by replicas and clients at each stage. In order to understand the normal execution of this protocol, while using digital signatures as a form of authentication, we follow the steps shown in Figure 2.10 :

1. A client sends a REQUEST message that consists of the operation requested, a timestamp, and the client's identifier, to the primary. If the client does not receive any reply, it then transmits the request to all replicas. In this situation, replicas forward the client request to the primary. A timer is also started by the client.

2. When the primary receives a REQUEST from a client, it assigns a view and sequence number, a digest of its history to an ORDER-REQ message, a digest of the client's message, and a collection of variables. The message is appended to the original client's message and sent to non-primaries.

3. When replicas receive ORDER-REQ messages, they verify the message's contents, specifically the request digest and sequence number. This message is then saved in their local history. Following that, replicas speculatively execute the request and forward it to the client in the form of a SPEQ-RESPONSE appended with the ORDER-REQ and result. The view and sequence number, a digest of its history, a digest of the result, a timestamp, and the client's identifier are all included in the SPEQ-RESPONSE message.

4. Client waits for a $3f+1$ matching replies. If it receives before the timeout, and everything matches up, the client commits to the reply.

2.3.3 Fault handling when having a faulty primary

Like any other protocol that utilizes a primary, a view change protocol is necessary when the primary acts faulty. Replicas only carry out a view change if they know that at least $f+1$ replicas will appoint a new primary. During this period, the process halts, and replicas only accept CHECKPOINT, VIEW-CHANGE, and NEW-VIEW messages.

Initially, when a replica detects that there is an inconsistency with the primary, it sends an "I-HATE-THE-PRIMARY" message for the current view. The replica continues to process requests while it does not receive at least $f+1$ "I-HATE-THE-PRIMARY" messages. Once it collects the required number of messages, it stops what it is doing in the current view and sends a VIEW-CHANGE message to all replicas. This message contains the next view number, a set of commit certificates, and the ordered request history. The set of commit certificates can contain the most recent commit certificate available, or $f+1$ VIEW-CONFIRM messages if there is no commit certificate, or lastly, a NEW-VIEW message if the other options are nonexistent. All of this information is crucial for when there are requests that end get committed at the replicas, but these do not know if the client received the amount needed of replies.

Upon gathering $2f+1$ VIEW-CHANGE messages, non-primaries set up a timer until they receive a NEW-VIEW message from the primary of the following view. If it expires, the process restarts, and replicas start a VIEW-CHANGE for the next view (v+2). After collecting VIEW-CHANGE messages, the new primary empties its base history. It proceeds to adopt either the most lengthened history validated by a commit certificate or $f+1$ matching histories, given that the former takes priority over the latter. It then transmits a NEW-VIEW message to all replicas. This message consists of the next view, a set of the

VIEW-CHANGE messages collected, and its own history(the copied history). In this set of VIEW-CHANGE messages sent in the NEW-VIEW, any request that shows that it got committed for at least $2f+1$ is known to be fully completed at the client. The former requisite is needed to have a safe history for the new view.

After replicas receive NEW-VIEW messages, they adopt the history provided by the new primary if valid. Finally, replicas multicast a VIEW-CONFIRM message acknowledging that they received a NEW-VIEW message. When the new primary receives $2f+1$ CONFIRM-VIEW, it can then start the new view and process requests.

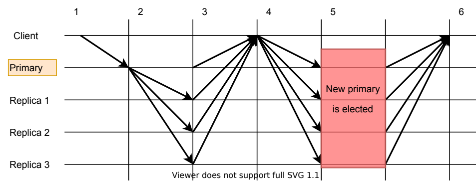


Figure 2.11: Zyzzyva execution with a faulty primary.

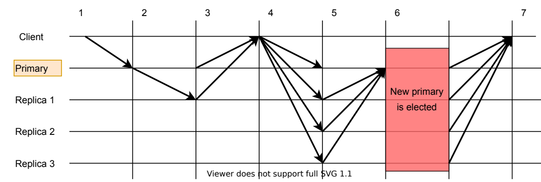


Figure 2.12: Zyzzyva execution with a faulty primary.

In the event that we have a faulty primary, and a client notices discrepancies in its responses, Figure 2.11 provide us the steps necessary to solve this issue:

2. A faulty primary sends different ORDER-REQ for the same request to all replicas.

3. Replicas speculatively execute the request and send a SPEQ-RESPONSE to the client.

4. Client receives diverse replies, triggering a Proof-of-Misbehaviour (POM) that is sent to all replicas. This proof essentially consists of ORDER-REQ messages that were dissimilar (for example, different sequence numbers or different histories).

5. As Replicas receive the POM, they verify that either the histories are different or the sequence number was different across the ORDER-REQ messages inside the POM. As such, they start a view change in order to elect a new primary.

6. The Client receives matching replies after the process has been restarted with a new primary.

In another scenario where client receives less than $2f+1$ replies, Figure 2.12 shows how the protocol mitigates the situation:

2. A faulty primary does not send ORDER-REQ message to all replicas.

3. Replicas that received ORDER-REQ progress to execute the request and send a SPEQ-RESPONSE to client speculatively.

4. The client's timer expires since it received less than $2f+1$. It proceeds to resend its request to all replicas.

5. All non-primaries send a CONFIRM-REQ, which has the request to be ordered by the primary, and the replicas itself start a timer.

6. There are two situations that can occur here, where in the first if the primary decides to delay the subsequent messages, it might trigger the timeouts on the non-primaries, which will then initiate a View Change and pick a new primary. As for the second, if the primary sends a CONFIRM-REQ, in which assigns a sequence number as well, to the replicas, these can then execute and reply to the client finishing the process.

7. Client receives matching replies after the process has been restarted with a new primary.

2.3.4 Fault handling when there is one faulty replicas

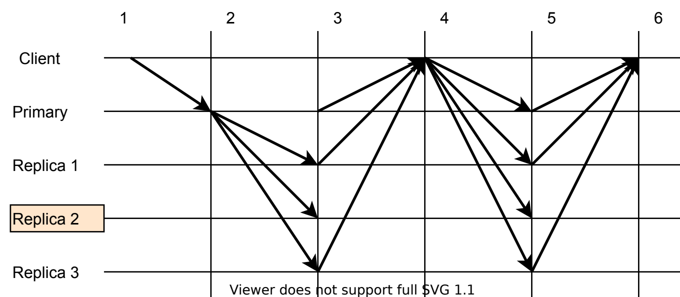


Figure 2.13: Zyzzyva execution when there is a faulty replica. [2]

In case the client's timer expires, and it only receives between $2f+1$ and 3f matching replies (Figure 2.13, from step 4 and on) :

4. The client progresses to create a COMMIT certificate that possesses SPEC-RESPONSE messages from replicas who replied and dispatches COMMIT messages to all replicas.

5. Replicas that receive COMMIT messages verify if their local history matches the history present on the message. If equivalent, replicas answer with a LOCAL-COMMIT to the client.

6. Upon receiving $2f+1$ replies, the client commits.

2.3.5 Fault recovery model

Replicas in Zyzzyva have a local log where they store the ordered requests they receive from the primary, called the ”history” log. They also store a copy of a max commit certificate, which is the commit certificate seen by the replicas with the highest sequence number.

Zyzzyva uses checkpoints that are built at each k-interval of requests. Each replica preserves a stable checkpoint and a tentative checkpoint. When a replica wants to generate a checkpoint, it starts by building a tentative version of it. It then multicasts a CHECKPOINT message. This message consists of the highest sequence number involved in the checkpoint, a snapshot, and a digest of the generated tentative checkpoint. To acknowledge that a checkpoint is stable, replicas must receive at least $f + 1$ matching CHECKPOINT messages and snapshots.

2.4 BFT-SMaRt

2.4.1 Protocol overview

BFT-SMaRt[10] [24] is a primary-based Byzantine Fault Tolerance Protocol which has a requirement of $3f + 1$ replicas in order to secure liveness and safety in presence of up to f faults. The communication model employed is partial synchrony. It uses MACs or digital signatures for message authentication, considering that authentication occurs in every protocol step.

This protocol has similarities with PBFT[9, 31] in that it employs a three-phase agreement. It uses a variety approaches that improve upon its predecessor, such as dynamic reconfiguration, which allows it to add/remove replicas on run-time and the usage of a different approach to state transfer, which enables replicas to reintegrate on a system without restarting the service. Most importantly, it implements a technique, denoted as VP-Consensus, to reduce the amount of communication steps between replicas through the use of abstractions that enable modular SMR implementations without the usage of reliable broadcast.

2.4.2 Fault Free execution

Figure 2.14 exhibits the usual pattern for fault-free execution. As for authentication, we are going to use MACs. This protocol consists of the following steps outlined below:

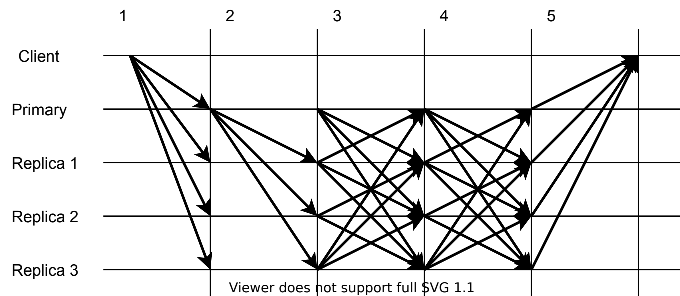


Figure 2.14: BFT-SMaRt normal execution. [24]

1. Client sends a REQUEST message to all replicas. This message consists of a sequence number used for once time semantics, the client's operation, the operation's type, and the client's identifier. It then proceeds to wait for matching replies with the same sequence number.

2. After receiving a REQUEST message, all replicas check to see if it contains a valid operation, sequence number, and MAC. The request is then stored in each replica's TO-ORDER set, and non-primaries start a timer. If this timer expires, it means that the primary did not propose anything. At this stage, the primary's role is to propose a batch of requests. As such, it sends a PROPOSE message to all non-primaries containing the sequence and view number and the batch of requests to be executed.

3. Upon receiving PROPOSE messages from the primary, replicas verify whether the sender is indeed the primary. If this is the case, non-primaries register the batch of requests that were proposed, and then send a WRITE message to all replicas, along with a digest of the proposed batch of requests. The WRITE message consists of the same information relegated by the primary, such as the sequence and view number and the batch of requests.

4. Upon receiving $2f+1$ WRITE messages with the same digest, replicas are decided and consequently produce a proof (certificate). This proof, as well as the batch of operations, are then recorded in the replica's log. The replicas then send an ACCEPT message to all replicas, which includes the decided-upon requests, the proof, and the view and sequence number.

5. Replicas who receive $2f+1$ matching ACCEPT messages proceed to execute the requests and send their results to the client via a REPLY message. This message

consists of the client's sequence number for once time semantics and the operation result.

6. The service is concluded when the client receives at least (2f+1) matching REPLY messages.

2.4.3 Fault handling with a faulty primary

BFT-SMaRt, like other protocols, possesses a View Change protocol. This protocol is known as "regency-change", so a view corresponds to a regency. For clarification, we will use the view change denomination. For instance, if we have a faulty primary that tries to starve the protocol by denying any REQUEST being proposed, a view change is started by the non-primaries if their timers expire twice:

1. A faulty primary after receiving a REQUEST from a client decides to not multicast a PROPOSE message to non-primaries present in the group. Non-primaries start the first timer after receiving the REQUEST message from the client.

2. When the first timer on non-primaries is triggered, they take action and multicast the request again to all replicas. Following that, non-primaries also start a second timer.

3. The faulty primary decides to ignore the retransmitted REQUEST messages. Non-primaries second timer is triggered, and a view change protocol starts. Replicas are forced to synchronize their states and appoint a new primary.

4. Replicas timers related to the other pending requests are aborted, and they multicast a STOP message (which includes the next view number and the set of requests that were time-outed) to elect a new primary.

5. Upon receiving more than f STOP messages, replicas start to change their view to a new one. Upon receiving more than $2f$ messages, the decision processing is interrupted, and the view is updated. Afterward, all replicas must be in the same state, so to achieve this, replicas send a STOPDATA message containing their history logs and the new view to the new primary.

6. After receiving STOPDATA messages, the new primary checks if there is no instance of the consensus missing. After collecting $n - f$ matching messages, it sends a SYNC

message to the non-primaries. This message contains information about the agreed-upon consensus instances to be followed.

7. To ensure that the primary sent correct information, non-primaries execute the exact computations performed by the primary after receiving the SYNC message from the primary. If the computations match, the replicas proceed to pick the same log (highest one). After these operations, they can return to resuming the previously paused decision processing and sync at the same state.

2.4.4 Fault handling with a faulty client

In case a faulty client decides to send a REQUEST message to all non-primaries but to the primary, the following steps (Figure 2.15) are taken by the system:

1. A client sends REQUEST messages to all non-primaries, but not to the primary.
2. Non-primaries receive the REQUEST and start a timer.
3. When the timer on non-primaries expire, they retransmit the REQUEST to the primary and start a second timer. In case this timer expires, a View Change (synchronization) phase begins.
4. The primary receives the REQUEST and begins proposing PROPOSE messages to non-primaries.

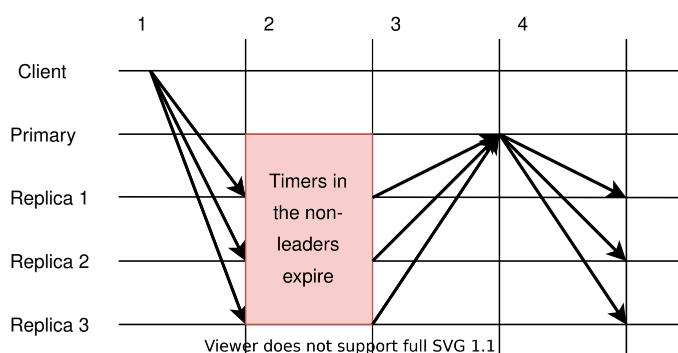


Figure 2.15: Execution with a faulty client.[24]

2.4.5 Fault recovery models

In regards to the recovery protocol, BFT-SMaRt uses a sequential checkpointing (Figure 2.16) protocol, where only a subset of replicas perform a checkpoint at any given time,

thus allowing the remaining replicas to process client requests without disrupting system availability. This protocol also improves the performance when the replicas take a snapshot of their state. The main advantage here is to have $n - f$ replicas continue processing requests while replicas take snapshots of their states at different times as each other. In parallel checkpointing (For instance, PBFT, Zyzyvva, and et cetera), there is performance degradation since all the replicas create a checkpoint simultaneously, halting the system from performing any actions during that time. So, in the end, replicas generate a checkpoint at their k-interval period, being that they do not construct a checkpoint at the same time as other replicas, avoiding the case that happens in parallel checkpointing.

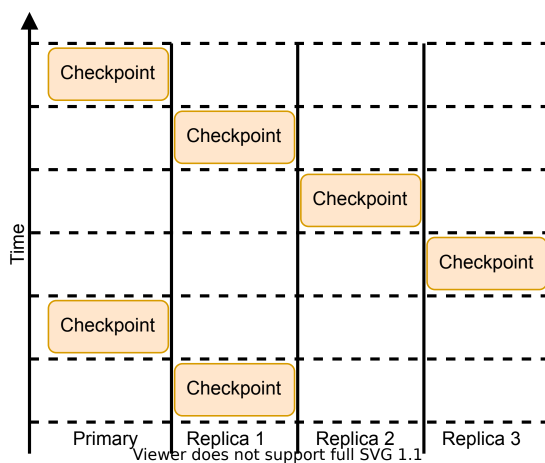


Figure 2.16: Sequential checkpointing. [33]

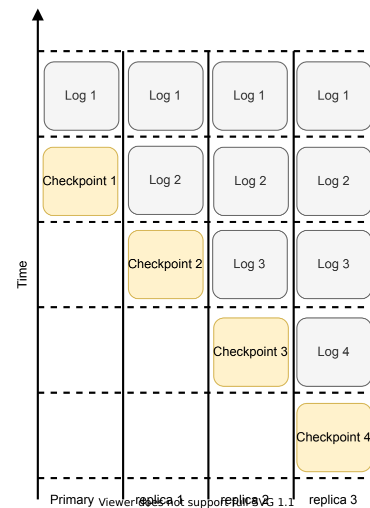


Figure 2.17: State transfer view of the protocol.[33]

Regarding state transfers, [33], this protocol is used to update states in replicas during recovery, using checkpoints and logs from replicas. There are several occasions that a state transfer can be triggered. For instance, if a replica is added during run-time, the replica will have to resort to this protocol to have its state updated to catch up to the other replicas states. Another example is when, after a view change phase, some replicas might have gaps in their decision log (its latest decision log does not match with another decision log). Furthermore, if a crashed replica is restarted and needs to update its state, the state transfer protocol is also started.

The state transfer protocol follows through several steps, where initially, a replica(or a leech) that needs to recover/update its state, checks its checkpoint, its local log, and dictates its agreement id(sequence number and last logged request). Then, the leecher proceeds to request the most recent agreement id from the seeders(the other replicas).

The seeders, after receiving a request, forward their agreement id's. Upon receiving $n-f$ agreement id replies from the seeders, the leecher obtains the state's id and the validation of these from the other seeders. Figure 2.17 shows how the segmentation of checkpoints and logs are distributed.

In this case, we will assume that the primary has the most recent checkpoint and the replica 3 has the oldest. Each log in the figure represents a partition of a replica's entire log. Each log partition is used as a validation to move a checkpoint's state to a more recent one. In order to validate, the leecher retrieves the states from the seeders, which contain the checkpoint and log partitions associated with it, as well as the corresponding hashes of log partitions and the most recent checkpoints from other seeders. The leecher has to replay the partitioned logs sent from the seeders so it can validate through the hashes and update its state. It then generates an intermediate checkpoint and computes the hash associated with it. Following that, it determines whether the hash of the intermediate checkpoint corresponds to the checkpoint provided by the replica. If they match up, it can safely stay in this state and proceed to the next checkpoint and use the partitioned logs associated with it. It terminates the state transfer protocol when it reaches the most recent state after several operations.

2.5 Prime

2.5.1 Protocol overview

Prime [14] is a primary-based Byzantine Fault-Tolerant Protocol which has a requirement of $3f+1$ replicas to ensure liveness and safety in the presence of up to f faults. The communication model used is partial synchrony. As concerns message authentication, it employs digital signatures(public-key cryptography) and uses a threshold signature protocol during a view change.

This protocol attempts to reduce the amount of degradation (time delay) that can occur with faulty replicas and, more importantly, the primary. In addition, it ensures that a primary will have to propose an ordering on all operations at some point without being unfair to non-primaries. It also attempts to adapt to network conditions and can determine an ideal time for replicas to timeout based on the latency between correct replicas.

Prime employs a novel ordering protocol in which non-primaries periodically multicast a summary message containing all the requests ordered that they know and force the

primary to send an ordering message containing the summary messages gathered to ensure undeniability and integrity. Non-primaries verify the performance of the primary regularly (for example, time taken by the primary to order requests to non-primaries), and in case the primary is slower than a certain threshold imposed, non-primaries vote to elect a new primary.

Non-primaries also frequently perform calculations to determine when they should expect an ordering message from the primary. This calculation is based on the primary's frequency of sending ordering requests, the latency between replicas, and a network-specific constant variable. To help with the monitorization of the primary, Prime allocates a fixed amount of work to be ordered. This method is used because the load that the system can handle is dynamic, which means that a primary could perform worse if there was a significant load of requests (similar to a denial of service), and non-primaries could believe that the primary was acting faulty, which would translate the system's state to a view change. If the load were static, there would be no need to allocate a fixed amount of work to the primary.

2.5.2 Fault free execution

As illustrated in Figure 2.18, this protocol has more steps than the previously discussed protocols. The following list illustrates how the system's normal execution is handled:

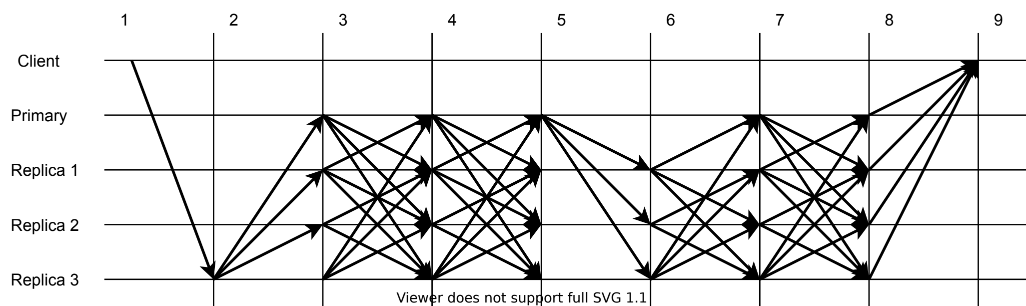


Figure 2.18: Prime normal execution. [14]

1. A Client sends a REQUEST message to one of the replicas at random. This message consists of the client's operation and a sequence number specific to that specific client, which increments each time that client makes a request. It starts a timer until it receives at least $f + 1$ matching responses from different replicas.

2. The replica that receives the REQUEST message validates the client's signature. It then assigns a preorder sequence number and sends a PO-REQUEST message to all replicas with the REQUEST details.

3. Once replicas receive PO-REQUEST messages, they verify the signature and the contents, specifically if this PO-REQUEST is new (they verify according to the preorder sequence number). If new, they multicast a PO-ACK message that includes a list of the replicas that sent a PO-REQUEST, a digest of the request, and the preorder sequence number.

4. Upon receiving at least $2f$ PO-ACK matching messages and verifying the signatures, replicas propagate a PO-SUMMARY message containing a preorder certificate, which proves that replicas have agreed on preordered requests and operations, as well as a list of replicas that sent a PO-REQUEST.

5. When the primary collects at least $2f$ PO-SUMMARY matching messages, it verifies the signatures and, knowing that replicas have requests preordered and are expecting ordering messages, it multicasts a PRE-PREPARE message. This message builds on the view and sequence number and a summary matrix. The matrix contains all of the entries from each replica's preorder summary. All replicas store the most recent PO-SUMMARY messages that they received from each other. If there is an inconsistent PO-SUMMARY message, it may indicate a faulty replica, which is then added to a blacklist.

6. Replicas upon receiving PRE-PREPARE messages, verify several elements. First, they check the signature, then if their preordered summaries match the primary's matrix preordered summary, if the view number is the current one, and whether it has processed that specific PRE-PREPARE message before. Following these checks, it multicasts a PREPARE message with a view and sequence number, as well as a digest of the preorder matrix summary from the primary. If replicas receive PRE-PREPARE messages from the primary with the same view number but different summary preorders, they blacklist the primary and call for a new primary election.

7. Upon receiving at least $2f$ matching PREPARE messages, replicas validate the signature. Then they execute the request and multicast a COMMIT message built on the PREPARE messages' information.

8. After receiving at least $2f+1$ COMMIT messages, replicas send a REPLY to the client. This message contains the client's sequence number as well as the operation's result.

9. The client commits to the request after receiving at least (f+1) matching REPLY messages. In case the timer expires before the client receives at least $f+1$ matching replies, the client proceeds to send the REQUEST to another replica or to at least $f+1$ replicas to ensure the request reaches a non-faulty replica.

2.5.3 Fault handling when having a faulty primary

In order to detect a possible faulty primary, Prime has a sub-protocol that verifies if a primary is acting suspicious called Suspect-Leader. According to Amir et al. [14], it employs three methods to achieve that goal. The first is that non-primaries are aware of what to expect when the primary sends PRE-PREPARE messages, primarily due to the former PO-SUMMARY messages that need to be included in the PRE-PREPARE message. The second is that non-primaries regularly evaluate how much time it usually takes the primary to send PRE-PREPARE messages (with their preorder summaries) after them sending their PO-SUMMARY message. Lastly, non-primaries can determine the speed at which the primary should be sending PREP-PREPARE messages. Through the use of the mentioned methods, replicas can measure the performance of the primary and detect whether the primary is faulty.

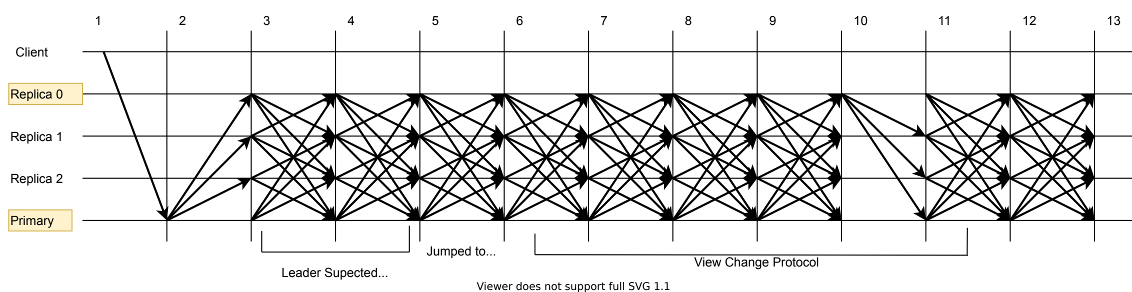


Figure 2.19: Prime execution under Leader suspect and view changing protocols.

If a replica detects that the primary of the current view is faulty, it takes action and sends a NEW-LEADER message to all replicas (containing the following view to which the replicas should jump) to change the primary.

The replica that sent the NEW-LEADER message continues to partake in the protocol until it collects $2f+1$ NEW-LEADER messages for a later view. Upon receiving $2f+1$

NEW-LEADER messages for the same later view, the replica that initially sent the message progresses to transmit a NEW-LEADER-PROOF message to all replicas and jumps to the following view. Replicas that received NEW-LEADER-PROOF messages jump to the next view, leave their current view and initiate a view change protocol.

The view change protocol attempts to address issues that PBFT [9] had with its view change protocol, most notably timeouts being doubled with each view change. So, they used a different approach where instead of the primary being the one coordinating everything, it only sends a REPLAY message with the information required for progressing the view change. Non-primaries can maintain a steady hand on the primary by checking if the received REPLAY messages are valid. This protocol uses a reliable broadcast protocol to guarantee that each replica is in the same state through dissemination. At the end of this protocol, a threshold signature protocol is used, so replicas can verify that the new primary is not misleading non-primaries.

Once a replica jumps to the next view, it initiates the view change protocol and multicasts a REPORT message (Starting from step 6). This message consists of the new view value, the last sequence number that a request was executed, and the size of its PC-SET(a set of prepared certificates for requests that the replicas sent a commit but did not get fully ordered). The prepared certificates in PC-SET are also sent afterward.

Any REPORT messages received by replicas are verified and logged. Replicas also prepare themselves to receive all the prepare certificates in the PC-SET of that replica. These are all logged after being received. After replicas collect $2f + 1$ REPORT messages and the prepared certificates in each PC-SET, they send a VC-LIST. This message consists of the view number and the replicas who sent the information. In the next step, after receiving VC-LIST messages, replicas progress to send a VC-PARTIAL-SIG message. This message builds on the information from VC-LIST, the sequence order that the primary will begin at, and a computed partial signature (threshold encryption) calculated into a tuple that includes the view, the list of replicas ids, and the sequence order of the primary. Replicas that receive $2f + 1$ PC-PARTIAL-SIG can then combine the partial signatures and generate a proof (VC-Proof) for the tuple of variables mentioned before. They then send a VC-PROOF message to all replicas containing the tuple's information and proof.

Following this, replicas begin monitoring the time it will take the primary to send a REPLAY message. As a result, the primary must send the message within a specific time frame before being suspected as faulty. After receiving VC-Proof messages, the primary

must send a REPLAY message (Step 10) containing all of the information gathered in the previous messages and define an ordering based on VC-PROOF (sequence order) and the most recent sequence numbered request that was executed (Found in REPORT). Following the previous step, it can then commit to using the state that the replicas sent to him. Finally, replicas that receive the REPLAY message attempt to agree via a REPLAY-PREPARE and REPLAY-COMMIT (similar to the standard execution protocol, so steps 11 and forward), once committed, they execute all agreed-upon requests in a batch.

2.5.4 Fault recovery model

In terms of the reconciliation protocol, replicas that have fallen behind with their PO-SUMMARY can be recovered. This process involves the need for $2f+1$ replicas to assist those who did not acknowledge some request preordering. This protocol employs MDS erasure coding, so each replica (up to the current state) sends a portion of the PO-REQUEST that is missing to the recovering replica. In this case, the PO-REQUEST is encoded in $2f+1$ parts, with only $f+1$ parts being required to decode it. After the decode, the recovering replica can update itself to the most recent state.

2.6 RBFT

2.6.1 Protocol overview

RBFT [15] is a primary-based Byzantine Fault Tolerance protocol that has a requirement of $3f+1$ replicas to ensure secure liveness and safety in the presence of up to f faults. It employs partial synchrony for its communication model. Message authentication is accomplished through a combination of digital signatures encapsulated in a MAC.

RBFT's approach attempts to mitigate issues related to primaries and what they are capable of when faulty, as it attempts to solve problems that previous robust protocols such as [12–14] had. This protocol uses multiple instances of the same execution running in parallel to compare the throughput of each other's instances, where each instance receives the same request from the client. There is one master instance that executes and replies to the client, while the other instances are referred backups (which monitor the master instance). Each instance contains one different primary in order to check if the master instance's throughput has degraded. If, by any chance, the master instance has a lower

throughput than one or more backup replicas, the primary in the master instance is likely to be faulty. When this occurs, a new primary is elected.

One of the most critical aspects of this protocol is the monitorization protocol in place in each node, as it monitors if the master instance has problems. It also measures the average latency for each request to be ordered. As for the clients, the primary in the master instance must be fair to all of them, or else an instance change would initiate due to backup instances monitorization. Each node contains a counter for each protocol instance. This counter has the number of requests ordered by each protocol instance. This counter is then used to compute the throughput of each instance replica on a regular basis before being reset. With this, it can compare its throughput with the master instance. In case the throughput is lower than a certain threshold, the backup instances will consider the primary of the master instance to be faulty and initiate an instance change.

2.6.2 Fault free execution

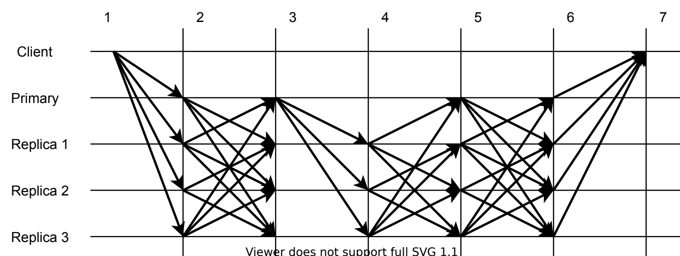


Figure 2.20: RBFT normal execution. [15]

Figure 2.20 shows the normal pattern of execution. This protocol works as the follow:

1. Initially, a client sends a REQUEST message to all replicas. This message is based on the operation requested as well as the clients and request identifier. This portion of the message is signed with the client's private key. A MAC is used to authenticate the entire REQUEST message.

2. Upon receiving a REQUEST message from the client, replicas check whether the MAC is valid or not. If valid, they validate the signature. If the signature is not valid, that client is added to a blacklist, meaning that any requests sent by that client will be ignored. If the signature is valid, replicas check if the request has already been executed. If this is the case, replicas resend the correspondent reply to the client. In case it is a new request, replicas send a PROPAGATE message that is multicasted.

This message contains the client's original request as well as a MAC. It is worth nothing that replicas will not send a PROPAGATE message if the other instance nodes did not receive the same request.

3. When at least $f + 1$ PROPOGATE messages are accepted for the same request, it indicates that the replicas are ready to begin processing the request. Following that, the primary sends a PRE-PREPARE message to all replicas, which includes a sequence and view number, as well as a digest and a MAC.

4. After receiving a PRE-PREPARE message from the primary, non-primaries proceed to verify the MAC. If valid, replicas store this message in their log. Next, non-primaries send a PREPARE message to all replicas only if their node has received at least $f + 1$ copies of the request, implying that the message will only be sent if the other instances receive it as well. This prevents a faulty primary and a faulty client from collaborating, in which the client would send valid requests to it, and that instance's performance would skyrocket in comparison to the others, which could lead to an instance change.

5. Upon obtaining at least $2f$ matching PREPARE messages, replicas verify if that the other replicas are consistent and prepare for commitment. This results in them sending a COMMIT message (consisting of the view and sequence number, digest, and the result) alongside a MAC to all replicas.

6. When replicas receive $2f + 1$ matching COMMIT messages, they can then reply to the client. Replicas subsequently send a REPLY message to the client. The message includes the result of the request execution done by the replicas alongside a MAC.

7. After receiving at least $f + 1$ matching REPLY messages from replicas, the client verifies the MAC and accepts the result of the protocol's request execution. The reply is only sent by the master instance.

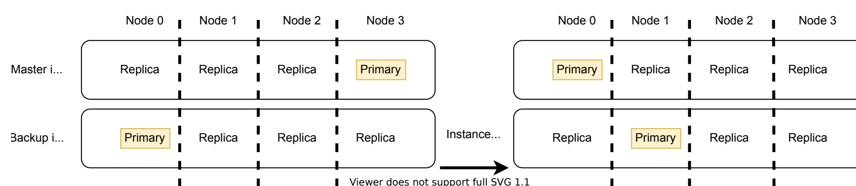


Figure 2.21: Instance change of RBFT. [15]

2.6.3 Fault handling when having a faulty primary

Because RBFT employs instances, changing the primary requires two protocols, the first of which is an instance change and the second of which is a view change. The instance change protocol, Figure 2.21, is used as a preparation to, later on, change the faulty primary in the master instance via a view change protocol. Each replica maintains a counter in order to identify each instance change message. When a node confirms that there is a significant contrast between its performance and that of the master instance, it can then assume that the primary in the master instance is faulty. With that, this node proceeds to send an INSTANCE CHANGE message (it sends its counter) alongside a MAC to the other nodes. Upon receiving that message, the nodes verify the MAC and if this message was for an older instance change. A comparison is performed to check if the counter received is smaller or greater than its own. If greater, then the message is discarded because it belongs to an older instance change. On the other hand, if the counter is smaller or equal, the nodes send an INSTANCE CHANGE message to all other nodes. Upon receiving $2f + 1$ matching messages, they initiate a view change in each protocol instance, meaning that each protocol instance gets a new primary and the faulty primary no longer is the primary on the master instance. They also increment the counter for each instance change.

2.7 Fault injection

One of the methods to test the systems described in this chapter, in real-world scenarios, is through the use of fault injection tools [34–36]. Fault injection attempts to determine whether the system's response meets its specifications and implementation, in presence of a defined range of different faults. Through the deliberate introduction of faults, fault injection tools enable both software and hardware developers to test and evaluate their system's reliability, consistency, performance availability, and security. In addition, these tools can also be used to perceive the effectiveness of fault tolerance mechanisms employed, such as recovery mechanisms, error detection, masking, and dynamic reconfiguration techniques.

The usage of fault injection tools has a number of advantages, according to Ziade et al.[37], including the following:

- The ability to study the behavior of systems in the presence of faults and their effects on the performance and functionality;
- The potential to find flaws in the design/implementation of systems;

- The ability to assess the effectiveness of fault tolerance mechanisms employed in systems such as recovery mechanisms, error detection, masking, and dynamic reconfiguration techniques;
- The prediction of the target system's faulty behavior, including a measurement of the efficiency provided by the fault tolerance mechanisms;
- The estimation of the failure coverage and latency of fault tolerant mechanisms.
- The ability to explore the effects that different workloads have on the effectiveness of fault tolerant techniques.

Several aspects categorize fault injectors based on the goal they want to accomplish. Such as how intrusive they are when injecting, how much control we have over them, the cost of assembly, the number of systems that can be used with them, the range of faults they have, whether they are used to test hardware, and so on. All of these characteristics are important in order for someone to choose and assess their system, considering that some of the aspects are more important as a factor of choice than others, such as, for instance, the types of faults to be used against the targeted system.

Fault injection encompasses a wide range of techniques, comprising typically into three categories, including software-based fault injectors, hardware-based fault injectors, and hybrid fault injector. Hardware injectors use gadgets to inject faults into hardware, being that these are capable of employing techniques like changing the temperature of the hardware[38], modifying voltages and electromagnetic interference's[39–41], port testing[42], pin-level testing[43, 44] and even the use of radiation[45]. Software injectors use programs/applications to introduce faults into software code during runtime or pre-runtime. Software-based injectors that rely upon runtime injection employ techniques that interrupt the execution of tasks while the system is running and inject faults during those periods[46–49]. As for pre-runtime injection (compile-time injection), faults are injected into the source code before the system is started.

Even though the approach of using software-based injectors is excellent and flexible, it has some limitations. One of these limitations is the inability to inject code into unreachable areas of software. Another flaw is that the injection of code may transform the original system's structure into a distinctive one that runs differently[37]. This possible outcome is primarily related to the level of intrusiveness, so software-based injectors must have a

relatively low level of intrusiveness in order not to disrupt/modify much of the original source of code.

2.7.1 BFT fault injectors

Fault injection can also be applied against systems that have been designed to tolerate Byzantine faults, as the ones that exist are software-based, such as Hermes[20], BFT-BENCH[22] and TWINS[21].

Hermes[20] is a fault injector that can inject faults in BFT-SMaRt[10, 24]. This fault injector allows users to assess the performance of BFT-SMaRt by injecting faults and observing the resulting behaviors. It uses an orchestrator to coordinate the various runtimes that are built into the replicas and client. The fault injection occurs through its runtime which is aggregated with either the replicas or clients of the protocol. It uses aspect oriented programming, specifically AspectJ, to make it easier to inject faults without changing the protocol's code. Hermes is also capable of injecting different faults in different replicas/-clients at the same time, meaning that each replica/client only runs one specific type of fault during each experiment. Lastly, this fault injector allows for the injection of a variety of faults, including network, crash, CPU load, and corrupted payload/headers.

BFT-BENCH[22], is a BFT fault injector capable of setting various environments and injecting faults in various deterministic BFT systems. BFT-Bench allows to define various execution scenarios and fault loads, as well as automatically deploy them in an online system and generate various metrics related to the monitorization. This allows for the analysis and comparison of the protocol's effectiveness in various situations. This fault injector has two useful features: a) BFT protocol selector, which launches a specific BFT implementation (PBFT, Chain and RBFT) with a specified number of replicas, and b) cluster setup, which allows clusters to be prepared to launch BFT protocols. BFT-BENCH employs an orchestrator that communicates directly with its "benchmark users" to prepare target replicas for evaluations and comparisons, and it employs some communication primitives to send commands to inject faults based on what was defined and collect metrics. To inject the faults, a daemon is run in the replicas to trigger the faults. This fault injector is capable of using a few faults such as, crash, delay, message flood, and increasing the number of clients that make requests in the protocol in a way to overload the system. It is worth noting that this injector can inject more than one fault at a time on each replica, but there's no way to specify how many rounds those faults run for, so they run indefinitely.

TWINS[21], is a recent fault injector which emulates byzantine behaviours by running two instances of the same replica/node with the exact specification and identity. Both replicated replicas/nodes are seen as one entity to the rest of the system, allowing to deploy different combinations of faults simultaneously. These replicated instances, for example, can send messages in the same consensus round, but they will contain contradictory information. Twins is made up of two critical components: the test executor and the test generator. The test executor is in charge of controlling message scheduling in accordance with a predefined scenario and ensuring that the system views each pair of replicas in each faulty node as one. The test generator, on the other hand, is responsible for creating test scenarios that are then sent to the test executor. Twins can generate various Byzantine behaviors such as, double voting, losing internal state, invalid messages and equivocation.

2.7.2 Summary of the BFT fault injectors

To summarize, all of these injectors take different approaches to fault injection, but some lack capabilities that a fault injector would benefit from, such as the use of faulty clients. The following table 2.22 shows a simple summary comparing the injectors briefly discussed with the fault injector developed throughout this dissertation, Zermia, which is further explained in this document's chapter 4.

	Fault Variety	**Faulty clients**	**Multiple fault triggering in the same replica/client**	**Can focus injection in one or more specific messages types**	**Multiple injection intervals in the same replica/client**	**Works in more than one BFT protocol**
Hermes	Medium	Capable	Not Capable	Not Capable	Not Capable	Not Capable*
Bft-Bench	Low	Not Capable	Capable	Capable(only one message type)	Not Capable	Capable
Twins	Medium	Not Capable	Capable	Unknown	Unknown	Capable
Zermia	High	Capable	Capable	Capable(one or multiple)	Capable	Not Capable*

Figure 2.22: Comparison of the characteristics between some of the existent BFT fault injectors.

The summary table 2.22 shows that many of these injectors lack some capabilities that could be used to develop more specific and edge case type scenarios, with Zermia being the exception. It is worth noting that both Hermes and Zermia have the potential to be adaptable to more protocols but lack the results to demonstrate it, particularly Zermia, which is a brand new fault injector that we did not have time to test with more than one protocol due to time constraints.

Chapter 3

Analysis of protocols under attack

In the previous chapter, we described some of the most relevant work in BFT and verified the trade-offs that each protocol faces. Unfortunately, these trade-offs may open up new attack vectors for a malicious actor to exploit in order to compromise the system.

As such, in this chapter, we discuss how a malicious actor can compromise the various protocols and how each protocol responds to different types of faults, as this will provide insight into what kind of faults to implement in tools that can assess these protocols, specifically fault injectors. Additionally, this will helps us to verify and prove that some theoretical attacks are feasible in practice against BFT-SMaRt[10, 24] later on.

3.1 PBFT

In this section, we explore what sort of fault environments PBFT[9, 31] fails to mitigate. For such, we investigate which sub-protocols fail to withstand fault scenarios not thought of, such as the agreement phase. We also look into what kind of faults can cause the system to slow down or even disrupt its state.

3.1.1 Protocol fault analysis

a) Agreement phase

As we learned from the previous chapter, PBFT clients send requests to all replicas in the form of $<REQUEST, o, t, c>_{\alpha_c}$. These requests are authenticated and logged on non-primaries, which causes a timer to start. When used maliciously by a client, some of the arguments in the REQUEST message can disrupt the system into starting a view change phase. Faulty clients may also choose not to send the REQUEST message to the primary

and instead send it only to the non-primaries, resulting in a view change. The following scenarios are identified in the following list:

1. A faulty client sends a REQUEST message to all non-primaries but the primary;
2. A faulty client sends a REQUEST message with the correct MAC to all non-primaries except the primary, which receives a REQUEST with a corrupted authenticator;
3. A faulty client sends a REQUEST message to all non-primaries except the primary, which receives a REQUEST with any or all of the arguments incorrect;
4. A faulty client sends a REQUEST message to each replica with different operations.
5. A faulty client sends a REQUEST message to each replica with a different timestamp;
6. A faulty client sends a REQUEST message to all non-primaries except for the primary, which receives a REQUEST with a timestamp inferior to a REQUEST previously executed for the same client;
7. The faulty client performs the same actions as in 4 and 5, but it only sends the REQUEST message to non-primaries.

The scenarios described above all have one thing in common, start a view change and replace the current primary.

In the first and last case, the primary cannot order requests since it does not receive them from the client (which also does not retransmit either). This situation eventually leads to non-primaries start a view change since they did not receive the request ordered by the primary before their timer associated with that request expired. Furthermore, because non-primaries in PBFT do not have request retransmission methods, the primary cannot do anything to avoid being replaced when the non-primaries timers expire.

In the second case, once the primary receives the REQUEST message, it attempts but fails to authenticate it. As a result, the primary discards the request. The non-primaries timer associated with that request expires, and a view change is initiated.

In the third, fourth, and fifth scenarios, the primary authenticates the REQUEST and sends a PRE-PREPARE message to all replicas. Once non-primaries receive the PRE-PREPARE, they compute the digest for the client request they have and compare them to the one sent in the PRE-PREPARE message, but they fail to do so. As a result, they initiate a view change.

In the sixth case, the primary discards the REQUEST since the timestamp for that message is inferior to one REQUEST previously executed for the same client. Once their timers go off, the non-primaries begin a view change.

We now investigate what happens in the remaining phases of the agreement protocol, where a faulty primary acting alone or colluding with a faulty non-primary can compromise consensus from the PRE-PREPARE phase to the COMMIT phase. As discussed in chapter 2, PBFT $<PRE-PREPARE, v, n, D(m)>_{\alpha_p}$ as well as $<PREPARE, v, n, D(m)>_{\alpha_i}$ messages have a similar arrangement, which includes the view and sequence number and the client's request digest. Both of these message's arguments can be altered to provoke a failure in the consensus during the COMMIT phase. As we know, the primary does not send PREPARE messages in the protocol, as this phase solely belongs to the non-primaries. However, a malicious primary can have the ability to send PREPARE messages if it so desires, and a non-primary may accept them. We assume that this case is possible since there is no indication to the contrary in the PBFT paper [9, 31]. The following list describes the possible scenarios in which consensus cannot be reached:

1. A faulty primary sends two separate PRE-PREPARE messages, each with a different sequence number. These messages are delivered to two different halves of the consensus group;

2. A faulty primary sends two separate PRE-PREPARE messages, each with a different view number. These messages are transmitted to two different halves of the consensus group;

3. A faulty primary sends two different PRE-PREPARE messages, one of which has the correct client request digest and the other does not. These messages are delivered to two different halves of the consensus group;

4. A faulty primary sends a correct PRE-PREPARE message to half of the group while transmitting a PRE-PREPARE with an incorrect MAC to the other half.

5. A faulty primary sends PRE-PREPARE messages to only $2f$ replicas and then stops participating in the protocol's later stages;

The scenarios portrayed share a common objective: to prevent consensus from occurring during the protocol's COMMIT phase.

In the first scenario, which can be accompanied through Figure 3.1, the faulty primary sends two PRE-PREPARE messages with different sequence numbers to various replicas. Those messages are sent so that only $2f$ replicas have the same sequence number. As a result, the faulty primary sends a PREPARE message with the same sequence number to replica 1, allowing it to generate a prepared certificate since it will have $2f$ matching(including its own) messages. Likewise, replicas 2 and 3 also produce a prepared certificate since they have $2f$ matching PREPARES, but for a different sequence number. Since the replicas are prepared, they send COMMIT messages, but the group cannot reach an agreement because the replicas have not received at least $2f+1$ matching COMMIT messages.

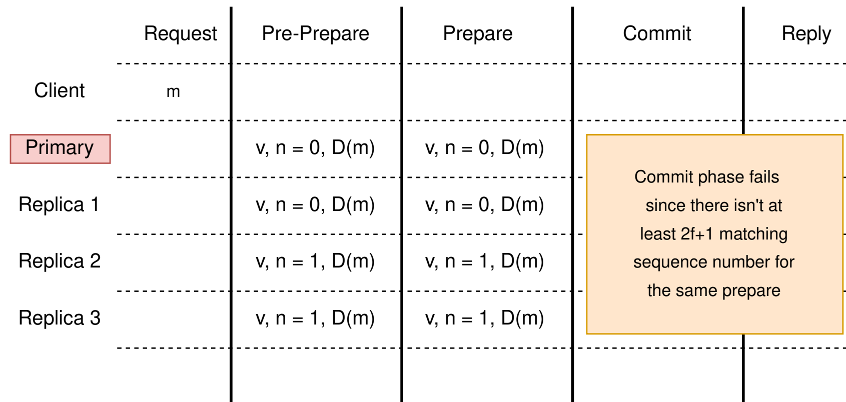


Figure 3.1: PBFT fail scenario 1.

In the second scenario, we repeat the steps from the first, but this time the view number is changed rather than the sequence number. This situation should result in the same outcome at the end, where an agreement is not reachable in the COMMIT phase.

In the third and fourth cases, one of the non-primaries receives a defective PRE-PREPARE message from the faulty primary and consequently discards it. The remaining non-primaries create a prepared certificate and enter the COMMIT phase. The faulty primary stops participating in this phase, and an agreement is never reached between the remaining two replicas. The other non-primary did nothing because it did not partake in the PREPARE phase as it discarded the defective PRE-PREPARE message sent by the primary.

The fifth case follows the same steps as the previous case, but with one minor difference. In this case, the faulty primary fails to send a PRE-PREPARE message to one of the non-primaries and stops participating in the protocol. This setup should also result in a non-agreement in the COMMIT phase.

b) View change phase

The view change protocol, on the other hand, cannot be exploited in any way known. We make this statement because of how the protocol is built, given that it leaves no room for a faulty non-primary to compromise the state since the minimum number of correct replicas to progress with the view change is well defined.

Since the consensus group needs $2f + 1$ messages to initiate a view change, non-primaries are not capable of using the message $<VIEW-CHANGE, v+1, h, C, P, Q, i>_{\alpha_i}$ for evil means. The same applies to $<VIEW-CHANGE-ACK, v+1, i, j, d>_{\mu_{ip}}$ messages, as faulty non-primaries cannot compromise the new primary with maliciously crafted messages considering the correct primary requires $2f$ matching VIEW-CHANGE-ACKS. Furthermore, during the NEW-VIEW phase, a faulty primary cannot compromise the state for non-primaries using crafted $<NEW-VIEW, v+1, V, X>_{\alpha_p}$ messages since non-primaries verify if the NEW-VIEW message received contains the former VIEW-CHANGE messages received by the replica in the V set.

3.1.2 Network and other faults

In PBFT, several faults can contribute to slow the system and thus decreasing performance. One example is delaying-based faults, which affect the majority of primary-based protocols. For instance, it is possible for a faulty primary delay the request ordering before replicas timer expires as stated by Y. Amir et al. [14]. Another exploitable scenario related to delays occurs during the view change phase. View changes in PBFT have a unique particularity in that they double the timeout on replicas every time a view change happens, but the protocol itself does not have a way to reset/reduce the timeouts. As a result, if a system experiences from multiple view changes across its lifetime, a faulty replica that becomes the primary can then further increase the delay to order requests.

This protocol is vulnerable to flood-based attacks, which involve faulty replicas flooding the system with messages (fake or correct), causing the correct replicas to waste time processing/filtering those. Additionally, due to the massive amount of illegitimate traffic passing by, replicas would struggle to progress with request processing in this scenario. These attacks could be amplified further by sending requests/messages of the largest possible size. On the client's side, there is a situation where misconfigured timers(related to them receiving replies) are set up to fire off almost instantly, forcing the clients resend their

requests every time that timer expires. This misconfiguration causes clients to flood the system and slowing it down since replicas need to verify/process any message they receive.

In the recovery aspect of PBFT, there are some issues with the proactive recovery protocol, which P. Sousa et al. [50] exposes. For example, the watchdog timer can be tampered with and reprogrammed to either delay or never have a chance to trigger. Another issue is that the recovery monitor can take significant time to finish its functions since it operates asynchronously. This mechanism, as P. Sousa et al. [50] points out, can be slowed down by delaying recovery messages in a hostile environment. The author also mentions the possibility of postponing recovery by slowing down local internal clocks.

3.2 Zyzzyva

For this segment, we examine at which fault scenarios Zyzzyva [2] fails to alleviate. As such, we investigate which sub-protocols have exploitable flaws as well as which faults can cause the system to slow down.

3.2.1 Protocol fault analysis

a) Agreement phase

As we can recall from the previous chapter, the agreement phase protocol in Zyzzyva does not appear to be vulnerable. For starters, we examine at what a faulty client can do in this protocol. Clients in Zyzzyva send requests in the form of $<REQUEST, o, t, c>_{\sigma_p}$ to one or more replicas. Faulty clients may attempt to induce a view change through crafted requests sent to different non-primaries. To understand how it is possible, we will go through the example below, where we will use two faulty clients, $c1$ and $c2$, to cause a view change.

1. Faulty client $c1$ sends three REQUESTS to all non-primaries with the same timestamp but different operations;

2. All non-primaries retransmit the REQUEST to the primary and start a timer associated with it.

3. The primary accepts the first REQUEST retransmitted from replica 3, orders it, and discards the remaining requests since they had the same timestamp.

4. Replica 1 and 2 timers expire and send an ”I-HATE-THE-PRIMARY” message to all replicas in the group, provided that this is insufficient to initiate a view change.

5. The request from $c1$ is speculatively executed and concluded.

6. Client $c2$ sends three REQUESTS to all non-primaries with the same timestamp but different operations.

7. All non-primaries retransmit the REQUEST to the primary and start a timer associated with it;

8. The primary accepts the first REQUEST with that timestamp and discards the remaining; however, in this case, it accepts the REQUEST from replica 1.

9. Replicas 2 and 3 timers expire and send an ”I-HATE-THE-PRIMARY” message to the entire group. This situation triggers a view change since all non-primaries have at least $2f+1$ ”I-HATE-THE-PRIMARY” messages to initiate a view change.

As we can observe, a view change occurs as a result of exploiting the ability of replicas timer expiring whenever the primary did not order their retransmitted requests and sending an ”I-HATE-THE-PRIMARY” message to all. Once replicas acquire $2f+1$ equal messages during the same view, they start a view change.

In the speculative phase, faulty primaries and non-primaries can only delay the protocol into a two-phase protocol by not replying to the client or sending a wrongful reply. If the faulty primary tries to send ORDER-REQ messages differently to each non-primary, the client, upon receiving the speculative SPEQ-RESPONSE, would verify the inconsistencies and send proof of misbehavior to initiate a view change.

b) View change phase

In recent work from I. Abraham et al. [51], the authors demonstrate that the view change protocol has some flaws that can of lead to the system jeopardizing safety through the collusion of faulty primary and faulty clients. The authors identify two instance in which Zyzzyva fails to guarantee safety. In the first fault scenario, the authors demonstrate that prioritizing COMMIT certificates over $f+1$ matching histories in the View change protocol might lead to situations not expected. The first scenario goes through three views at $F=1$, which are as follows:

1. In the first view, two faulty clients($c1$ and $c2$) send their requests($Rc1$ and $Rc2$) to the primary. The faulty primary sends ORDER-REQ messages for $Rc1$ to the non-primaries 2 and 3 and transmits an incorrect ORDER-REQ to non-primary 4, which contains $Rc2$. Following that, both the faulty primary and non-primaries 2 and 3 speculatively execute $Rc1$ and send a SPEQ-RESPONSE to $c1$. Since $c1$ only collects $2f+1$ matching SPEQ-RESPONSE's, it generates a commit certificate and proceeds to transmit a COMMIT message for $Rc1$ to only the faulty primary due to the client either being faulty or because there is a significant network delay. The certificate is saved by the faulty primary.

2. The system enters a view change (view 2) caused by either network delay or the faulty primary provoking it. In this view, the new primary(replica 2) gathers $<VIEW-CHANGE, v+1, CC, O, i>_{\sigma_i}$ messages from itself, replicas 1(faulty) and 4. The VIEW-CHANGE message from itself contains its history execution for $Rc1$, whereas the other two VIEW-CHANGE messages contain the history execution for $Rc2$ from replica 4 and faulty replica 1 (which sends for $Rc2$ rather than $Rc1$). The new primary then adopts a new history consisting of $Rc2$, since there were at least $f+1$ matching histories in the $2f+1$ VIEW-CHANGE messages received. The primary broadcasts $<NEW-VIEW, v+1, P>_{\sigma_p}$ messages and all replicas adopt the primary's history and clear their own. Considering that the history of the primary and other replicas contains $Rc2$, they all speculatively execute it and send a SPEQ-RESPONSE to $c2$). The client receives $3f+1$ responses, and all replicas store the committed $Rc2$ in the first position in their history.

3. The system goes through another view change (view 3) caused by a network delay. The new primary (replica 3) collects VIEW-CHANGE messages of itself and non-primaries 1 and 4. The new primary uses its history, and replica 4 sends its history, given that both should have the same committed request ($Rc2$) saved. However, the faulty non-primary sends a commit certificate for $Rc1$ in the VIEW-CHANGE message. The new primary opts for the commit certificate for $Rc1$ rather than the $f+1$ histories, builds its history based on it, and sends NEW-VIEW messages to other replicas. The non-primaries adopt the primary's new history, which clears their own history. They then execute $Rc1$ speculatively, send SPEQ-RESPONSE to $c1$, and replicas save the committed $Rc1$ in the first position in their history.

The second fault scenario focuses on how the View change protocol's prioritization of the most extensive commit certificate does not always work as intended. Like the previous case, it starts with a faulty primary, goes through three views, but uses two-phase protocols and four clients to break safety. The explanation for this scenario is the following:

1. In the first view, client $c1$ through client $c4$ send different requests, where $Ac1$ and $Ac2$ matches the first two clients and $Bc3$ and $Bc4$ correspond to the remaining clients. The faulty primary sends two ORDER-REQ messages for $Ac1$ and $Ac2$ to non-primaries 2 and 3, where the first request is posted in the first place in their histories and the second request is placed sequentially after the first. The non-primary 4 receives two incorrect requests from the faulty primary that contain $Bc3$ and $Bc4$ and places them in the correct order in its history. Both the primary and non-primaries execute their requests speculatively and send them to their respective clients. Client $c2$ only receives $2f + 1$ and progresses to create a commit certificate with replicas histories which contain $Ac1$ and $Ac2$. Due to network delays, the client sends a COMMIT message with the commit certificate imbued to only replica 3.

2. The system enters a view change (view 2), caused by either network delay or a faulty primary provoking it. In this view, the new primary (replica 2) gathers VIEW-CHANGE messages from itself and non-primaries 1 (faulty) and 4. The VIEW-CHANGE message from itself consists of its execution history for $Ac1$ and $Ac2$, whereas the other two VIEW-CHANGE messages contain execution histories for $Bc3$ and $Bc4$, which are from replica 4 and faulty replica 1(sends for $Bc3$ and $Bc4$ instead of $Ac1$ and $Ac2$). The new primary then adopts a new history consisting of $Bc3$ and $Bc4$, since there were at least $f + 1$ matching histories in the $2f + 1$ VIEW-CHANGE messages received. The primary broadcasts NEW-VIEW messages, and all replicas adopt the primary's history by clearing their own. Both the primary and non-primary speculatively execute $Bc3$ and send a SPEQ-RESPONSE to client $c3$. Since the client does not receive $2f + 1$ matching replies, it generates a commit-certificate for $Bc3$ (only for this request since $Bc4$ is yet to be executed). The client then sends a COMMIT message to all replicas containing the newly created commit certificate. Once replicas receive the COMMIT message, they send a LOCAL-COMMIT message to the client. The client receives $2f + 1$ replies, and all replicas save the committed $Bc3$ in the first position in their history.

3. The system goes through another view change (view 3) caused by a network delay. The new primary (replica 3) collects VIEW-CHANGE messages of itself and from non-primaries 1 and 4. The primary uses its commit certificate that was stored in the first view for $Ac1$ and $Ac2$. Replica 4 transmits the commit certificate for $Bc3$ and its history consisting of $Ac1$ and $Ac2$. As for the faulty replica 1, it can either match what the primary or replica 4 are sending or send an empty history. Since the protocol dictates that the primary must choose the most extended commit certificate, it proceeds to pick its own commit certificate as it has two requests ($Ac1$ and $Ac2$) versus the commit certificate replica 4 (or the faulty replica) sent, which had only one($Bc3$). The primary builds its history based on the commit certificate picked and sends NEW-VIEW messages to other replicas. The non-primaries adopt the new history from the primary, which clears their own history. Replicas then speculatively execute ($Ac1$ and send a SPEQ-RESPONSE to the client $c1$. This committed request(($Ac1$) is then stored in the first place at each replica's history.

In more recent work, I. Abraham et al. [52] solves the problems mentioned above by changing how primaries and non-primaries select safe histories from the primary during view change.

3.2.2 Network and other faults

The implementation of Zyzzyva dictates that for a speculative execution (one-phase protocol) occur correctly, $3f + 1$ matching replies are needed to be received at a client. In the event that several replicas fail, the system may be forced to operate with $2f + 1$ replicas. This scenario implies that the protocol would have to constantly go through a two-phase protocol in which the client accepts $2f + 1$ matching replies during the second phase.

A faulty primary can amplify the previous scenario by initially ignoring a client's request. This client will then have to retransmit the request since it did not receive replies to all replicas. The non-primaries retransmit the request to the primary and begin a timer. The faulty primary takes advantage of that and delays the transmission of ORDER-REQ and sends to only $2f$ replicas. The faulty primary and the $2f$ replicas that received ORDER-REQ send a SPEQ-RESPONSE to the client. Because the client only received $2f + 1$ matching replies, the protocol goes into two phases. Because of this setup, the system takes a significant time to process requests.

In Zyzzyva, there are some ways to flood the system with messages and therefore slow down the system. For instance, a faulty non-primary can flood the primary with FILL-HOLE requests. This situation can occur if a faulty replica when receiving ORDER-REQ from the primary reports that the sequence number of this request is greater than expected. As a result, the primary will have to keep sending ORDER-REQ messages to fill alleged ”holes” in the faulty replica history. The previous scenario can be further amplified to the point that a faulty non-primary continues sending FILL-HOLE messages to all replicas, which then they have to send ORDER-REQ for every received FILL-HOLE message, slowing down the system further. Faulty clients can also cause similar floods if their retransmission timers are set up to go off immediately, which leads them to transmit their requests to all replicas endlessly. A Faulty primary can also delay ORDER-REQ's from FILL-HOLE requests from a non-primary because non-primaries start a timer related to the reception of that message.

3.3 BFT-SMaRt

Through this section, we examine BFT-SMaRt[10, 24] and its sub-protocols by verifying which are and are not compromisable. We begin by inspecting the agreement and view change protocol, and afterward, we analyze some attacks that are proficient in delaying the system by abusing the primary.

3.3.1 Protocol fault analysis

a) Agreement phase

Clients in BFT-SMaRt send requests to replicas in the consensus group in the form of a $<REQUEST, nextSeq, op, readOnly>_{\alpha_c}$. Since we know how the request is built, we can now ascertain what a faulty client can do or not and what measures the protocol has to mitigate potential attacks.

We begin by verifying what happens when a faulty client sends requests to different replicas, where each request has different operations but the same sequence numbers. Two possible scenarios can emerge from this setup.

In the first case, the primary proposes its request, starts its timer, and waits to receive messages for the next phase while discarding retransmitted requests from non-primaries

(with the same sequence number from the same client but different operation). The primary's timer expires and disseminates the request, so other replicas have it in the ToOrder set. After this step, it can safely propose, and consensus may progress since there are at least $2f + 1$ replicas with knowledge of that request.

As for the second scenario, once non-primaries receive the request, they save it in the ToOrder set and start a timer. Meanwhile, once the primary gathers the request from the client, it orders it. The non-primaries receive the proposal but discard it, as they do not have it in their ToOrder set. After some time, all replicas timers expire for the first time and consequently disseminate their request in the ToOrder set to all other replicas. There are two possible outcomes: the primary adopts one of the retransmitted requests or ignores all of them because it already has a request in its ToOrder set with that sequence number. In either case, the replica's timer would expire again because they had not received the pending request in their ToOrder set, resulting in a view change.

If a faulty client sends a request to multiple replicas, each with a unique sequence number, each of these requests will be retransmitted across all replicas in the consensus group once the timers associated with that request expire. Because each request is considered unique based on the sequence number, the primary ends up ordering the requests received from the non-primaries, and consensus can progress if there are conditions, such as (2f+1) replicas having the request in the ToOrder set.

During the $<PROPOSE, Cid, Ep, i, val>_{\alpha_i}$ phase, it is the responsibility of the primary to propose requests for execution in the group, given that those requests must be already in the ToOrder set in the non-primaries for them to accept. A faulty primary may attempt to modify the PROPOSE message in order to change the consensus id (sequence number), the Epoch (view number), or even the batch of requests to propose (val) and send to different non-primaries. These setups would be rejected once non-primaries received the malicious PROPOSE's, because if the sequence or view number differed from what was expected, non-primaries will suspect the primary and initiate a view change. Suppose the faulty primary sends a different batch of requests, and the non-primaries do not have some of them in the ToOrder set. In that case, they discard the message and proceed to a view change if the timers associated with the requests yet to be ordered expire twice (the first timer is related to the request retransmission).

Both $<WRITE, Cid, Ep, i, v>_{\alpha_i}$ and $<ACCEPT, Cid, Ep, i, v>_{\alpha_i}$ phases are not compromisable by a faulty replica since the consensus group requires $2f + 1$ correct replicas to

progress. As a result, faulty replicas are incapable of crafting malicious messages or failing to send them in order to disrupt the consensus.

b) View change phase

In terms of the view change protocol, faulty replicas cannot initiate a view change on their own because the protocol requires replicas to receive a minimum of $2f+1$ $<STOP, reg, M>_{\alpha_i}$ messages in order to begin the primary replacement. Therefore, changing the view (reg) or requests that were not ordered in the STOP message provides nothing significant to confuse correct replicas and start a view change. The phase afterward, $<STOPDATA, reg, DecLog>_{\alpha_i}$, follows the same principle, given that faulty replicas are incapable of compromising the state by changing the view or the history log in the message since the primary requires to gather $2f+1$ matching STOPDATA's to progress. Finally, in the $<SYNC, creg, Proof>_{\alpha_p}$ phase, a faulty primary can try to send different SYNC messages, but will result in a new view change because non-primaries verify the contents of the SYNC message and perform the exact computations as the primary. As a result, any discrepancy would result in those replicas sending STOP messages to initiate a view change.

3.3.2 Network and other faults

Like many primary-based focused protocols, BFT-SMaRt suffers from faulty primaries that delay request ordering. A faulty primary can delay PROPOSE messages right before the timer on the non-primaries expire, preventing a view change in the process. Faulty primaries can delay the request process of ordering further by ignoring the client's request initially and wait for the retransmission of that request from the non-primaries. It then progresses to delay the PROPOSE message before the timers on the non-primaries expire and with that prevent a possible view change.

One of the many benefits of BFT-SMaRt is the ability to add/remove and recover replicas in real-time. The ability of this protocol allows for the possibility of an attack based on repeatedly crashing a faulty replica and initiating a state transfer out of it. Correct replicas that attempt to restore the state of the faulty replica to the current one may be slowed while this process takes place. Another scenario involving state transfers is when a faulty replica (a leecher) reduces system throughput by indefinitely requesting state transfers from the other replicas (seeders). For example, suppose a faulty replica is constantly "trying" to get to the same state as other replicas. In that case, it may reduce

performance because one replica is always recovering, and the others are attempting to support the leecher.

By default, the authentication of client's requests is performed by using MACs [53]. This definition may lead us to employ a strategy similar to that used in PBFT. The client can send a MAC vector that gets authenticated at the non-primaries with the exception of the primary. This situation can cause a view change since the primary will not propose a request that it cannot authenticate, leading the non-primaries to distrust the primary since it is not proposing. To avoid this, BFT-SMaRt can opt to use digital signatures rather than MACs.

3.4 Prime

As discussed in the previous chapter, Prime[14] tries to solve problems associated with faulty primaries through the use of multiple mitigation techniques. Through this segment, we examine what sub-protocols are compromisable and which are not, while also explaining why they are exploitable. We begin by examining its agreement and view change protocols, and then we verify some attacks that are capable of delaying the system by exploiting the monitorization aspect of each replica.

3.4.1 Protocol fault analysis

a) Agreement phase

We begin by determining what a faulty client can do to destabilize the protocol. Requests sent by clients are in the form of $<CLIENT-OP,o,seq,c>_{sigma_c}$, which are transmitted to one or more replicas. A faulty client lacks the means to disrupt the protocol by sending malformed or differently crafted messages to different replicas. The reason is that if the faulty client decides to send, for instance, two requests with varying identifiers, both of them would be ordered since replicas would treat them as separate requests due to the unique identifier. The faulty client could also try sending requests with different operations, but only the first request received by replicas would be ordered since the other requests have the same unique identifier.

During the preordering phase, the replica that receives the request from the client has the function to distribute it amongst its peers. It sends a $<PO-REQUEST,seq_i,o,i>_{\sigma_j}$ to

inform other replicas about the client's requests. A faulty replica can attempt to send different PO-REQUEST messages to distinct replicas with either different sequence numbers or operations, but that would result in nothing. This assertion is based on the assumption that once other replicas receive it, they would send $< PO - ACK, i, seq_i, D(o), j>_{\sigma_j}$, and after receiving PO-ACK's from other replicas, they would notice that the PO-REQUEST and at least $2f$ PO-ACK messages cannot be compared or do not match between themselves in order to build a preorder certificate. Afterward, during the $<PO - SUMMARY, vec, i>_{\sigma_j}$ phase, a faulty replica can try to send a crafted message with different vector values; however, after sending two inconsistent messages, other replicas would detect the discrepancy and blacklist that replica. Furthermore, replicas in this phase must receive $2f + 1$ matching PO-SUMMARY messages to progress, meaning that faulty replicas cannot disrupt this phase.

As for the global ordering phase, a faulty primary might try to send invalid, or even $<PRE - PREPARE, v, seq, sm, l >_{\sigma_l}$ with different matrix summaries. This setup would not work because the non-primaries perceive and compare the summaries they have (along with those received from other replicas), which would mean that the primary could be subject to a replacement. Both the PREPARE and COMMIT phases are not disruptable by faulty replicas as the protocol has a well-defined number of replicas to tolerate both phases.

b) View change phase

In relation to the suspect leader protocol, faulty replicas are incapable of provoking view changes since replicas in the consensus group only start transitioning to a new view when they receive $2f + 1$ $<NEW - LEADER, v + 1, i>_{\sigma_i}$. The same applies to $<NEW - LEADER - PROOF, v + 1, S, i>_{\sigma_i}$ messages, as the faulty replica is ineffective at disturbing the group, as they need as well $2f + 1$ messages to preinstall the new view correctly. After this step, the actual view change protocol starts.

After the suspect leader protocol finishes and replicas preinstall the new view, the view change protocol starts. Afterward, replicas send a $<REPORT, v, execARU, numSeq, i>_{\sigma_i}$ to all its peers. A faulty replica could try to modify some of the arguments in their REPORT message, such as the number of prepare certificates it has(for example), and match them with fake prepare certificates sent after in various <PC-SET, v, pc, i>$_{\sigma_i}$ messages. This setup would be impossible since the correct replica would verify that those certificates are inconsistent with what it has in its log. Also, this plan would end up not affecting the

system since replicas need to collect $2f + 1$ REPORT messages, meaning that the messages sent from the faulty replica, when compared, would have an irregular state in relation to the other REPORT's received from correct replicas.

The remaining phases, such as VC-LIST, VC-PARTIAL-SIG, and VC-PROOF, cannot be disrupted as they always require $2f + 1$ correct replicas. Once replicas send the $<VC - PROOF, v, ids, startSeq, p, i>_{\sigma_i}$, they expect a REPLAY message to be received by them based on a timer started by the replicas based on the turnaround time. A faulty primary is capable of delaying the $<REPLAY, v, ids, startSeq, p, l>_{\sigma_p}$ message by tiny amounts based on what should be the turnaround time for correct replicas to receive that message. A faulty primary can also try and modify the REPLAY message and send it to different replicas. However, this would be later confirmed as inconsistent since non-primaries broadcast a REPLAY-PREPARE to all their peers and compare messages REPLAY-PREPARES between them. This situation would eventually lead up to a new view change.

3.4.2 Network and other faults

According to Aublin et al.[15], Prime has flaws despite its mitigation techniques for faulty primaries. The first flaw detected by the authors is the ability of non-primaries to fail their monitoring and produce imprecise reports that anticipate requests arriving later than usual. A faulty primary can take advantage of this weakness and cause message ordering to be delayed for longer than usual.

Prime is capable of being exploited in another way, according to Aublin et al.[15], where a faulty client colludes with a faulty primary to increase the round-trip time. In order to trigger this flaw, a faulty client needs to send a request of the maximum size possible while requests of smaller sizes are being either sent or executed. Through this approach, the round-trip increases, and non-primaries doing the monitoring are tricked. The faulty primary can then delay ordering requests further based on the maximum round-trip time values.

3.5 RBFT

As we have seen in the previous chapter, RBFT [15] employs a robust architecture that tries to mitigate every kind of fault scenario possible. In this section, we investigate if it is possible to compromise the protocol, and if not, explain why it should not be achievable. We begin by reviewing its agreement protocol and view/instance change. In the later part

of the section, we check what faults do not compromise the system state but can, for instance, slow down the agreement protocol.

3.5.1 Protocol fault analysis

a) Agreement phase

A client in RBFT sends requests in the form of $<REQUEST, o, rid, c>_{\sigma_c}, c>_{\mu_c}$ to replicas to process it. From a faulty client's perspective, it does not possess means of attacking the protocol through request forging, corrupting requests, or even flooding. We can make this claim because RBFT has several mitigation techniques for dealing with such attacks. The following list verifies some of the possible scenarios for faulty client attacks and the protocol's countermeasures.

- If a faulty client sends a request with an incorrect/corrupted MAC or signature, correct nodes will verify that the client is untrustworthy and stop receiving requests from that client while adding the client to a blacklist.
- Flood attacks from faulty clients are insignificant because each node has a NIC (Network Interface Controllers) to guard against those initiatives.
- A faulty client may attempt to confuse nodes by sending different requests with different parameters. This attack fails because a node only sends the client request to its $f + 1$ replicas if it receives $2f + 1$ matching requests.

During the PROPAGATE phase, since we recognize that the nodes receive at least $2f + 1$ matching client requests before sending them to their respective replicas, we know that it should be difficult for any faulty replica/node to disrupt the system with fake requests attached to the $<PROPAGATE, <REQUEST, o, s, c>_{\sigma_c}, i>_{\mu_i}$ message. We make this assertion because replicas receive at least $2f + 1$ matching PROPAGATE messages independently of what the faulty node/replica does.

In terms of the PRE-PREPARE phase, a faulty primary cannot confuse the system by sending conflicting $<PRE-PREPARE, v, n, c, rid, d>_{\mu_p}$ messages with altered arguments such as the sequence, view number or even its MAC. The reason for this is that the primary is correct in the backup instance ($F = 1$), which would always order the exact requests correctly as in the main instance. On the other hand, the faulty primary would end up suspected because once replicas received the PRE-PREPARE, the nodes corresponding to

those replicas would compare and verify it had the same PRE-PREPARE in both $f+1$ instances. If this does not happen, the nodes would not be able to send PREPARE messages, resulting in a view change. Furthermore, because of the monitorization and comparison between instances, a faulty primary cannot significantly delay any ordering of requests.

The remaining phases, PREPARE, COMMIT, and REPLY, can withstand the attempts from a faulty replica (whether primary or not) to disrupt the group since they would have enough correct replicas to reach a consensus and reply to the client.

b) View change phase

As for the view change, faulty nodes cannot provoke a view change by themselves when sending $<INSTANCE_CHANGE, cpi_i, i>_{\mu_i}$ messages. Faulty nodes may attempt to send an incorrect number for the counter or even try to start the instance change. Of course, these efforts will be futile since replicas requires to receive $2f+1$ matching INSTANCE-CHANGE messages.

3.5.2 Network and other faults

According to Gupta et al. [22], in RBFT, when the primary in the main instance crashes, the process for changing the primary does not get invoked. When an event of this nature occurs, clients in this protocol rebroadcast requests to all replicas. This situation triggers a continuous flood-based attack from a portion of the clients. The authors claim that this error is caused by clients lacking crash handling mechanisms.

As stated by the authors of RBFT [15], some of the worst attacks are capable of deteriorating the protocol's performance, albeit insignificantly. The first attack goes as follows: a) A client sends REQUEST messages to all replicas with the exception of the primary replica in the master instance. B) Following that, faulty replicas flood the primary with invalid PROPAGATE messages of the maximum size. C) Afterward, the faulty replicas of the master instance overwhelm the correct replicas with invalid messages of the maximum possible size. D) Finally, faulty replicas of the master instance do not participate in the rest of the protocol. With all of this in place, the system barely suffers any repercussions, which can be observed in the results in the paper of P.-L. Aublin et al. [15].

Another attack involving a faulty replica in the master instance is also mentioned by the authors. Where initially, a faulty client sends invalid REQUEST messages to the correct

replicas. Subsequently, faulty replicas flood the correct replicas with invalid messages of the maximum possible size and proceed to not partake in the PROPAGATE phase. Finally, the faulty replicas in all backup instances flood the correct replicas with invalid messages of the maximum possible size, and they, like before, do not participate in the rest of the protocol.

3.6 Summary

	Message Delaying	Message Manipulation	Replica Crashing	Flood Attacks
PBFT	X	X	X	X
Zyzzyva	X	X	X	X
BFT-SMaRt	X	X	X	X
Prime	X	X	X	
RBFT		X	X	

Table 3.1: General summary of the vulnerabilities found in each protocol.

Table 3.1 summarizes the vulnerabilities discovered during the analysis of each protocol in this chapter. Most of the protocols depicted here show weaknesses regarding delaying-based faults, as this is essentially due to a faulty primary's ability to slow request ordering. RBFT is the only protocol that mitigates this issue with efficacy, as it was built with this issue in mind. Prime can also mitigate this attack effectively, except in some instances where faulty clients collude with faulty primaries or when non-primaries fail the monitoring. Message manipulation, such as altering the inner contents of messages, changing the MAC (for some protocols only), altering the message transmission target, applies to all protocols in some of the cases. As concerns replica crashing, all protocols are in some way affected, purely on the fact that a faulty primary can purposely crash to initiate a view change and that can be somewhat costly in terms of performance. Another aspect about this fault is that through a faulty primary crash it is possible to enable attacks against messages in the view change protocol that would not be accessible under normal circumstances. Lastly, PBFT, Zyzzyva, and BFT-SMaRt protocols are prone to flood attacks from faulty clients, primaries, and potentially non-primaries, as they do not have the means of mitigation that Prime and RBFT have regarding this type of attack.

It is worthy to note that while some protocols here are capable of mitigating more faults than others, these were designed specifically to mitigate those types of issues while sacrificing performance, latency or throughput to achieve those goals.

Chapter 4

Zermia: A fault injector for BFT systems

In the course of this chapter, we introduce Zermia, a new fault injector capable of injecting faults in BFT systems. First, in section 4.1, we present Zermia's primary architecture. Then, in section 4.2, we present how Zermia's main components work through their lifecycle. Finally, throughout section 4.3, we describe Zermia's implementation details, as well as some other features.

4.1 Zermia's architecture

Zermia is a fault injector based on a client-server model, where a server acts as a coordinator for the runtime Agents (clients) who run alongside the replicas/clients of a targeted protocol, as presented in Figure 4.1.

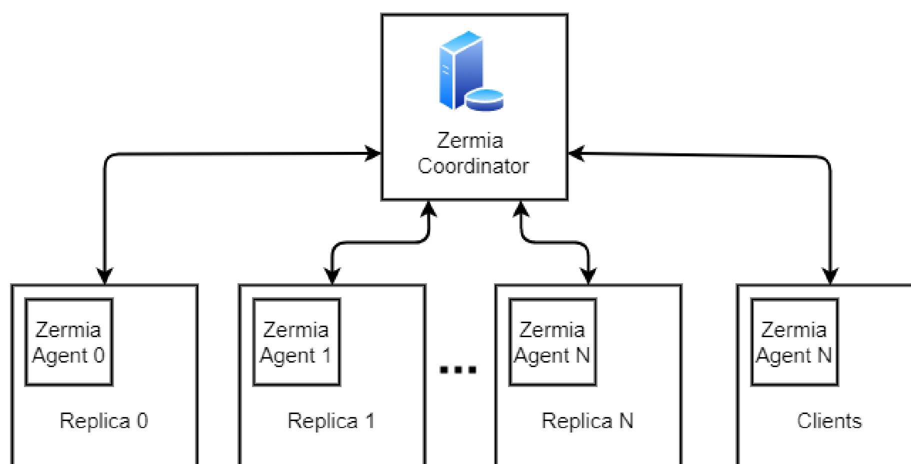


Figure 4.1: Overview of the Zermia Architecture

4.1.1 Coordinator

The Coordinator is responsible for ensuring that Agents comply with the fault schedule defined by the user. It is also in charge of distributing fault schedules to Agents, collecting metrics and essential data, and displaying it at the conclusion of each experiment (if the option is enabled).

4.1.2 Agent

The Agent executes in the same execution environment as the target application and is responsible for requesting information from the Coordinator and injecting faults in the correct order according to the fault schedule provided by the Coordinator.

Agents attach to specific points of the target program and alter the target's execution state and flow, by accessing method calls, arguments, and return values. In addition, Agents maintain additional states, independent of the application, such as the number of iterations of the application's main loop or, in our case, the number of consensus rounds. This solution expands Zermia's flexibility by allowing it to use this knowledge to trigger faults at predetermined intervals.

4.1.3 Faults

Several factors characterize faults in Zermia, including what the fault is and what it does, where to inject the fault (for instance, inject into specific messages), when to inject the fault (the starting point of injection), how many times the fault triggers, and lastly, which parameters the fault has and which are customizable.

By design, Zermia provides a set of predefined faults that can be parameterized and scheduled. These involve communication and network-related faults typical in the BFT context, such as, for instance, message delay, message modification, and message drop. Later in this chapter, we go over the general faults that this fault injector can use.

4.1.4 Fault schedules

In Zermia, fault schedules are a sequence of one or more faults that will be injected into replicas and clients during a test run. These schedules are created by users based on a variety of factors, including:

- Which replicas or clients that will be faulty;

- How many faults are declared in each schedule;
- Which faults and their parameters to be used in each schedule phase;
- The initial triggering point for each fault;
- The number of times to trigger each fault;
- Which types of messages to inject the fault;
- Who should faulty clients/replicas target their attack against (for example, attack only the primary).

4.2 Zermia's model and test cycle

As previously stated, in Zermia, the system's key components are the coordination server it hosts and the runtime Agents aggregated to the replicas/clients of the protocol. Both of these components go through three phases during their test cycle: initial configurations, booting sequence, and execution procedure.

4.2.1 Initial configurations

a) Coordinator Side

In order for the Coordination server to function correctly, some configurations related to the targeted protocol system must be made ahead of time. At a minimum, the coordinator must be aware of the following:

- How many clients and replicas are available in total;
- How many replicas partake in the consensus;
- How many clients partake in the experiment;
- Which are the ID's (identification) of each replica and client.

For the purpose of building fault schedules, the Coordinator also needs to know which parameters were introduced by the user when starting the server, as these are the following:

- Which replicas are faulty (if any);
- Which clients are faulty (if any);

- Which faults were assigned;
- What are the parameters chosen for each fault;
- What is the starting point of injection of each fault;
- How many iterations each fault is going to trigger for;
- Which type of messages are to be targeted with the injection;
- Who should faulty clients or replicas target their attacks against, for instance, the entire consensus group or only the primary.

In addition, the Coordinator must configure two essential elements that concern itself: the port that will be open for communication and the length of time it will be online for.

b) Agent Side

On the Agent side, very few configurations need to be reviewed, which primarily concerns establishing a communication channel with the Coordinator and metric purposes. As such, each Agent must know the following:

- The Coordinator IP address;
- The Coordinator port number open for the service;
- The number of consensus rounds at which the process of gathering metrics stops;

4.2.2 Booting process

a) Coordinator Side

Once the configuration is completed, the Coordinator can be initiated. Initially, the Coordinator starts by acquiring information on the properties of the clients and replicas involved in the experiment, such as their IDs and the number of participants for each entity. This data is then used to lay the groundwork for the database of the clients and replicas.

Afterward, the Coordinator retrieves the parameters from the fault schedule introduced by the user and reviews it for possible typos, errors, or if the starting rounds for each fault are ordered correctly from the lowest to the highest. This information is then used to populate the remaining fields on the replicas and client databases.

Lastly, the Coordinator configures and starts its services to receive and transmit messages to any Agent. The services start on the port that the user has previously set in the configurations.

b) Agent Side

After the Agents setup are complete, and the Coordinator is up and running, the target protocol replicas and clients can be initiated. Once clients/replicas boot, the Agents associated with them are started shortly after. The Agents then collect information from their host, such as their ID, and gather information required to establish a communication channel later with the Coordinator.

4.2.3 Execution process

a) Coordinator Side

Once the Coordinator starts its services, it is ready to receive messages from the Agents and respond accordingly. The general services provided by the Coordinator are the following:

1. AgentStatus - The Agent queries the Coordinator to know if the client or replica it is associated with is to be faulty or correct;
2. RequestScheduler - The Agent queries the Coordinator for the corresponding fault schedule associated with that client or replica;
3. UpdateMetrics - The Agent sends the Coordinator the metrics it gathered throughout the experiment for the client or replica is associated with.

Whenever the Coordinator receives a request concerning the Agent's status (AgentStatus), it checks the database using the information provided in the request to see if the client/replica with whom the Agent is associated is to be faulty or correct. If the client or replica is to be faulty, the Coordinator replies to the Agent with that status and initial information regarding the fault schedule it will execute. If the client or replica status is non-faulty, the Coordinator replies to the Agent with that status only.

When the Coordinator is queried with a scheduling request (RequestScheduler) from an Agent that has its status as faulty, the Coordinator verifies the corresponding database (client or replica) to see if that Agent has faults to run. If there are faults to run, the

Coordinator sends the corresponding fault schedule over one or more messages, which depends on the size of the schedule itself. The message contains which faults to run, their parameters, and the period of time they are to be triggered.

Whenever the Coordinator receives a message containing test metrics (UpdateMetrics) from an Agent, the Coordinator performs simple calculations to determine the average throughput and latency of the replica to which the Agent is associated. Once the Coordinator receives all replica's end-of-test results, it outputs all information, including the elapsed test time, the total number of messages, and average throughput/latency for each replica. It is worth noting that in this service, the Coordinator does not reply to the Agent.

b) Agent side

After a successful boot sequence, the Agents use the previously gathered information on the replica or client associated (such as the ID) and transmit it to the Coordinator (through an AgentStatus request). Once the Agents receive the Coordinator's reply, they update their status (correct or faulty) based on the message's content and store the remaining information related to their fault scheduling on them. If the Agent's state is to be correct, all future instructions are skipped with the exception of metric-related functions.

In the event that the Agent's state becomes faulty, the Agent sends one or more fault scheduling requests (FaultScheduler messages) to determine which faults to execute, with the number of requests sent determined by the size of the fault schedule. As soon the Agent receives the Coordinator's replies, it stores the message's contents, including which faults to run, the fault parameters, the starting trigger point for each fault, and the number of times to trigger each fault.

After knowing its fault schedule, the Agents halt their execution till specific functions are invoked on the replica's or client's side. These are interrupted by Agents to either gather information about arguments, alter them or run faults.

Once replicas or clients reach a starting trigger point for one or more faults (at a certain consensus round), the Agents wait until a specific function is called to interrupt and then inject the faults according to schedule. The faults are then triggered for one or more rounds depending on the schedule. It is worth noting that the Agent passes the control to its host once faults are finished triggering at each instance of consensus round.

As soon the Agent concludes triggering all faults in one of the intervals of time, it verifies if there are more faults to trigger at later time intervals. If so, it waits till the client

or replica hits the next starting point of injection. This process continues until there are no more faults to be injected in other time intervals.

When the test is finished, the Agent transmits all of the experiment's information (UpdateMetrics message), such as the total number of messages and elapsed time to the Coordinator. It should be noted that the Agent does not receive a reply from the Coordinator.

4.3 Zermia's Implementation details

Zermia's implementation is currently written in Java and integrates Agents with replicas and clients using AspectJ (Aspect-Oriented programming). Communication between Coordinator and Agent is accomplished through the use of remote procedure calls, specifically gRPC [23]. Zermia's current implementation is at present time only integrated with BFT-SMaRt [10, 24].

4.3.1 Coordinator

a) Fault schedule gathering

We used the command line to receive data from the user in order for the Coordinator to be able to acquire information about the fault schedule. The commands entered adhere to a set of rules, where we first must digit which replicas will be faulty, then which faults to inject, their starting trigger point, and finally the number of times they are triggered. As for clients, these have an exception where we only need to specify the number of faulty clients once, and all of the clients will follow the same fault schedule. Figure 4.2 presents an example of a fault schedule for two different replicas and five faulty clients:

```
java -jar Zermia.jar Replica 0 TdelayAll 100 5000 10000 Replica 2 TdelayAll 100 5000 10000
Crash 1 20000 1 Clients 5 PNRS 500 100
```

Figure 4.2: Example of a fault schedule through command line

In this example, replica 0 injects a fault that delays processing for 100 milliseconds whenever it intends to send a message where the starting trigger point is at the consensus round 5000, and the fault is then triggered for 10000 rounds, ending at round 15000. As for replica 2, a similar scenario happens in the first interval of time, where the same fault runs at the same starting round and for the same length. Once replica 2 reaches the consensus round 20000, a crash fault is injected within the same round, disabling that

replica afterward. As seen with replica 2, the starting round needs to be in ascending order, or else the application won't boot it and show an error. Concerning the five faulty clients, once they reach the sequence number of 500 (starting round), they stop sending messages to the primary but continue to send to the non-primaries for 100 rounds.

b) gRPC services

```
service ZermiaCoordinatorServices{
        rpc AgentStatus {..}
        rpc RequestScheduler {..}
        rpc UpdateMetrics {..}
}
```

Figure 4.3: Coordinator service interface (gRPC methods)

The Coordinator uses gRPC as means to communicate with each Agent and distribute their corresponding fault schedules. This technique serves as a groundwork to later expand and adapt the injector to other languages. Figure 4.3 reviews the general services previously explained in the former section.

```
message AgentStatusRequest{
        string replicaID {..}
}

message AgentStatusReply{
        bool replicaStatus {..}
        int32 faultScheduleSize {..}
        int32 groupSize {..}
        repeated string faultyReplicaList {..}
        repeated bool messageTypes {..}
}
```

Figure 4.4: Agent status requisition service details

As previously explained, the Agent's status service, Figure 4.4, is the first request an Agent makes to the Coordinator. The Coordinator after verifying for the status of associated host through its ID in the database, it replies to the Agent with the status (faulty or correct), the system's group size, the list of faulty replicas (for faulty clients, this last section is excluded), and finally, the list of message types that are a target of the injection for that Agent. We send all this information only if the host is supposed to be faulty, by contrast, if non-faulty the reply is sent only with the status. It is worth noting that the list of faulty replicas provides means of faulty replicas not attacking each other and, the size of the fault schedule is used as an iterator in the service of FaultScheduler.

Figure 4.5 presents the service for fault requesting, where the Coordinator replies to the Agent with the arguments of their fault schedule if existent. The Coordinator uses the schedule iterator on the Agent's request to look through the fault schedule database

```
message FaultSchedulerRequest{
        string replicaID {..}
        int32 faultScheduleIterator {..}
}

message FaultSchedulerReply{
    repeated string faultParameters {..}
    repeated string faultTriggerParameters {..}
}
```

Figure 4.5: Fault schedule requesting service

associated with that host and consequently sends a reply containing that part of the fault schedule. The faultParameters field contains a list of the fault and its parameter. The faultTriggerParameters field contains the starting trigger point and the number of times to trigger the faults.

```
message UpdateMetricsRequest{
        string replicaID {..}
        double testFinalTime {..}
        int32  testTotalMessages {..}
}
```

Figure 4.6: Update metrics service details

As for the last service, UpdateMetrics, Figure 4.6, the Coordinator stores all metrics received from the Agents, such as the elapsed test time and the total number of messages, and calculates the latency and throughput for each replica.

```
INFO: ------------- Replica 0 END OF TEST STATS --------------
Replica 0 END TIME : 332.83 seconds
Replica 0 THROUGHPUT : 260.0 messages per second
Replica 0 total messages : 86643 messages
Replica 0 average latency: 3.8 millisenconds
Oct 17, 2020 2:31:14 PM zermia.proto.services.ZermiaRuntimeServices statsService
INFO: ------------- Replica 1 END OF TEST STATS --------------
Replica 1 END TIME : 332.89 seconds
Replica 1 THROUGHPUT : 173.0 messages per second
Replica 1 total messages : 57446 messages
Replica 1 average latency: 5.8 millisenconds
Oct 17, 2020 2:31:14 PM zermia.proto.services.ZermiaRuntimeServices statsService
INFO: ------------- Replica 2 END OF TEST STATS --------------
Replica 2 END TIME : 332.83 seconds
Replica 2 THROUGHPUT : 172.0 messages per second
Replica 2 total messages : 57356 messages
Replica 2 average latency: 5.8 millisenconds
Oct 17, 2020 2:31:14 PM zermia.proto.services.ZermiaRuntimeServices statsService
INFO: ------------- Replica 3 END OF TEST STATS --------------
Replica 3 END TIME : 332.83 seconds
Replica 3 THROUGHPUT : 174.0 messages per second
Replica 3 total messages : 58035 messages
Replica 3 average latency: 5.7 millisenconds
```

Figure 4.7: Test results demonstrated in the Coordinator.

Once all replicas finish the experiment and transmit their metrics, the Coordinator displays all data in the terminal, as presented in Figure 4.7. Additionally, all the displayed information is also saved in an excel file. This service only applies to replicas and not clients, as there was no metrics from faulty clients relevant to end results.

4.3.2 Agent

Through the use of advices, agents interact with replicas and clients. Advices are code blocks that are inserted into the target application's code at runtime via join-points. Join-points are the areas of code in the target application where advices are inserted. These have semantics for specifying whether an advice executes before, after, or around the respective join-point, which is referred to as pointcuts. The around semantics can also prevent the corresponding join-point from being executed. When a client or replica starts its main function, an agent is initiated via an advice that is associated to that method.

Figure 4.8 presents the initial advice that is used to start the Agent whenever a replica's main function is started in BFT-SMaRt. Within this aspect class, an Agent thread is instantiated and associated with the replica's runtime where it remains active as long the replica does not crash or ends its functions. As for clients, Agents are initiated before they send their first request message to replicas, as their main function does not provide information relevant to initiate communication with the Coordinator.

```
@Aspect
public class ZermiaInstanceStart {
    (...)
    @Before("execution method_main")
    public void advice(JoinPoint jp) {
        String[] HostArgs = (String[]) joinPoint.getArgs()[0]; //get host arguments
        String HostID = HostArgs[0]; //get replica/client id
        ZermiaAgent.getInstance().setID(HostID);  //save host id in the agent
        try {
           ZermiaAgent.getInstance().InstanceStart(); //Agent boot & status gRPC request
           ZermiaAgent.getInstance().faultScheduler(); //fault schedule gRPC request
        } catch (Exception e) {
           e.printStackTrace();
        }
    }
(...)
}
```

Figure 4.8: Advice function which is called whenever a replica's main function is started in BFT-SMaRt.

When a replica or client reaches the code blocks that manage message sending, one or more different advices are called for the management of fault injection. The current implementation that we have with BFT-SMaRt contains several advices that are used for the injection of faults. Figure 4.9 presents the general idea of the main fault injection advice that is called whenever a replica or client tries to send a message. The idea here is that if an Agent is faulty and the current consensus round lies between the starting and ending trigger point for a fault in the schedule, that fault is then triggered. Once that fault ends triggering completely, the iterator for the fault schedule moves forward for the subsequent fault with its associated starting and ending trigger points.

```
public class ZermiaFaultManagement {
    int faultStartingTriggerPoint; //starting point for a part of the schedule
    int faultEndingTriggerPoint; //ending point for a part of the schedule
    int currentConsensusRound;
    int scheduleIterator;
    string faultName;
    (...)
    @Around ("execution method_send")
    public void execute(JointPoint jp) {
     (...)
        if(AgentStatus.equals(true)){ //if faulty
            if(faultStartingTriggerPoint <= currentConsensusRound &&
            faultEndingTriggerPoint > currentConsensusRound){
                switch(faultName){
                    case"fault1":{
                    (...) //fault method
                    break;}
                    (...)
                    case"faultN":{
                    (...) //fault method
                    break;}
                }
            if(faultEndingTriggerPoint == currentConsensusRound){
                scheduleIterator=+1;
                //remaining code here updates the variables associated to the schedule
                (...)
                }
                (...)
            }
        } jp.proceed();
    }
    (...)
}
```

Figure 4.9: General idea of the fault injection advice.

Agents use additional advices to keep track of the state of its host, such as the view they are at, the current round of consensus, who is the primary of the group, and so on. This gives some flexibility to decide how to trigger the faults.

As for the client, the Agent follows a similar path to inject faults as for the replicas. Once clients reach their send function, an interruption is forced by the aspect class to follow the fault schedule sent from the Coordinator. In this advice, the Agent performs two jobs: one is the fault injection and the other the capability of changing targets to send messages at will (either the primary, non-primaries or the entire system).

Is worth noting that due to limitations to what we can do with aspects, very few functions that we interrupt can not be used to entirely alter arguments, create messages and inject faults without changing that same interrupted function. As such, we changed some functions while maintaining the normal functioning to include the faults inside of it.

4.3.3 Predefined faults in Zermia

Zermia's current implementation includes a standard set of network, communication, and protocol-specific faults for testing BFT systems. Each fault can be configured to increase

or decrease its strength based on the needs of the user. The following list provides an overview as well as specifics about the faults that Zermia is capable of injecting:

- Crash - Crashes the client or replica by executing a System.exit() method.
- Delay Thread - Delays processing time through the use of Thread.sleep() method. The user can configure the delay in milliseconds. This fault occurs several times per consensus round based on the number of function calls invoked to send messages.
- Message Dropping - Prevents messages from being sent to other replicas/clients by preventing the target method call. This fault can be executed based on a probability (0 - 100%) defined beforehand by the user of the fault to occur.
- Message Flood - Sends a variable number of messages by encompassing the target method in a loop. The number of loop iterations, which translates into the number of additional messages sent, can be configured by the user. It can also be combined with the fault described below, message modification, to increase fault flexibility.
- Message modification - Replaces message contents by modifying argument values passed to the target join-point. The message's contents can be defined by the user and can be fully empty or even fully randomized.
- Resource exhaustion - Exhausts the application's resources by employing algorithms that can increase CPU usage, decrease available memory RAM, or even cause a crash due to memory RAM depletion. The user can specify the number of threads for the CPU load and the amount of memory that should be leaked.

4.3.4 Schedule conflicts

In the case that multiple faults occur within the same interval of time, the user can set or disable a priority system in advance based on their preferences. If enabled, some faults will be triggered before others during the injection, maximizing the fault's impact on the system. If this option is disabled, the system will run the faults in the user's order and specifications. Through this method, the user is granted more options to produce more effective fault schedules if desired.

For instance, if the priority system is enabled and a delay thread and a crash fault occur within the same time interval, the delay thread fault will be triggered first, followed

by the crash after the specified delay. This example shows how to prevent a crash without first exploiting another fault.

Chapter 5

Experimental Evaluation

In this chapter, we present the evaluation of Zermia by displaying its versatility and capability. To this end, we designed different fault schedules that target BFT-SMaRt[10] protocol which was reviewed in chapter 2. To assess the system in question, we used the YCSB[54, 55] client/server already implemented in the BFT-SMaRt library, as it presents metrics that allow us to view the influence in performance each fault schedule has on the target library.

5.1 Experimental Setup

Each test branches into two setups based on the number of replicas present on the system, which were for four replicas($F = 1$) and ten replicas($F = 3$). All experiments consisted in running YCSB with 50 clients for a total of 2.5 million operations (approximately 300000 consensus rounds). Experiments were measured based on the system's throughput in operations per second.

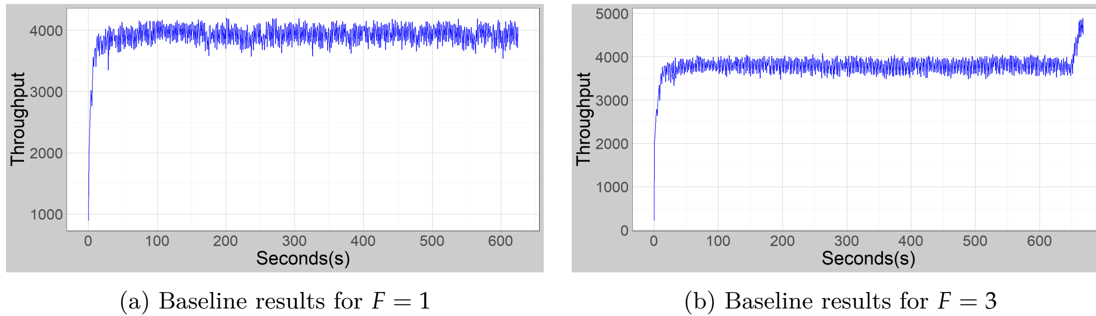
Fault triggering started in consensus round number 50000 for $F = 1$, and rounds 100000 and 150000 for $F = 3$. Each fault was triggered for a variable number of rounds based on its type. The presented results in the following sections are the average from the results of 5 consecutive runs.

Regarding BFT-SMaRt configurations, we enabled the following cipher suite ”TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256” as it was needed in order for the protocol to work. The number of replicas and faulty replicas is also adjusted based on the test cases. The remaining configurations stayed as the default.

For test evaluation, we used a total of twelve virtual machines from Google Cloud services where we used: 3-10 for the YCSB BFT-SMaRt replicas, 1 for the YCSB BFT-SMaRt client, and 1 for the Zermia Coordinator. Each virtual machine ran Ubuntu 18.04.5 LTS operative system, with 16 gigabytes of RAM, 4-core AMD EPYC 7B12 clocked at 2250 MHz, and 10 gigabytes of HDD.

5.2 Baseline results

The following illustrations, 5.1a and 5.1b, show us the results obtained from systems where no faults were injected. The outcomes display a similar performance to each other despite the increase in the number of replicas. These results serve as a way to compare with experiments where we injected faults.



(a) Baseline results for $F = 1$

(b) Baseline results for $F = 3$

Figure 5.1: Results from a fault-free experiment

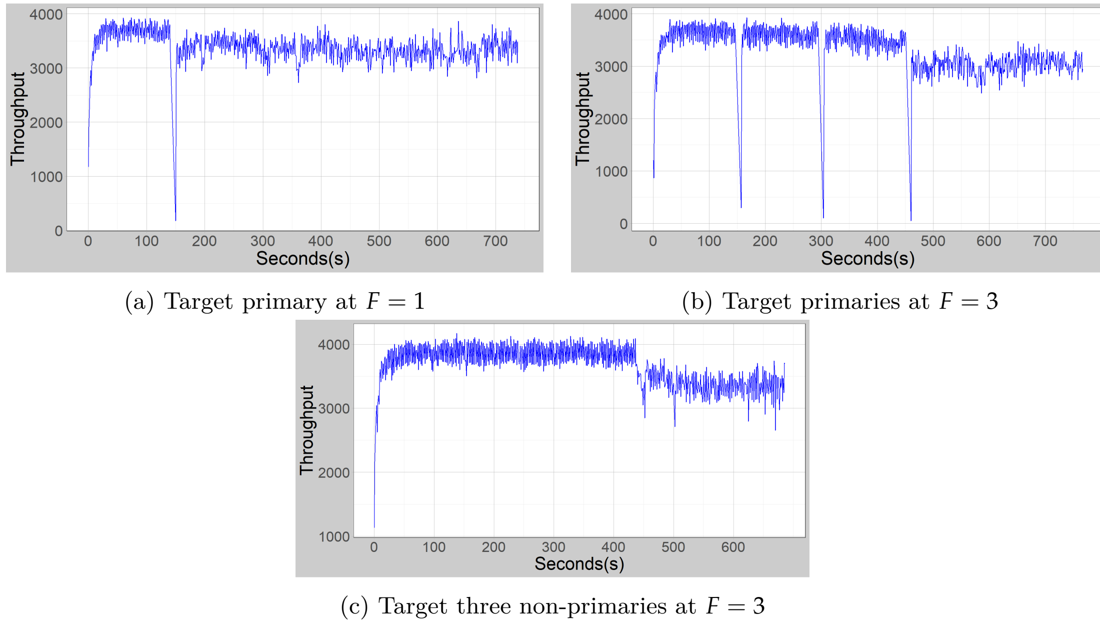
5.3 Communication-based faults

5.3.1 Crashing Replicas

This experiment consisted in injecting and executing a fault schedule that crashes faulty primaries and non-primaries. Faults were triggered at the consensus round number 50000 for $F = 1$, and 100000 and 150000 for the $F = 3$ configuration. In each of these starting points, the fault was triggered for a total of 50000 consensus rounds.

Figure 5.2 presents the results when injecting a crash fault against the primary replica for $F = 1$ (Figure 5.2a) and $F = 3$ (Figure 5.2b) configurations, and for non-primaries at a $F = 3$ (Figure 5.2c) configuration.

From what we can gather from the outcomes in Figures5.2a and 5.2b, we can verify that a drop in performance always occurs whenever a crash occurs in a primary, which



(a) Target primary at $F = 1$

(b) Target primaries at $F = 3$

(c) Target three non-primaries at $F = 3$

Figure 5.2: Results for a crash fault schedule

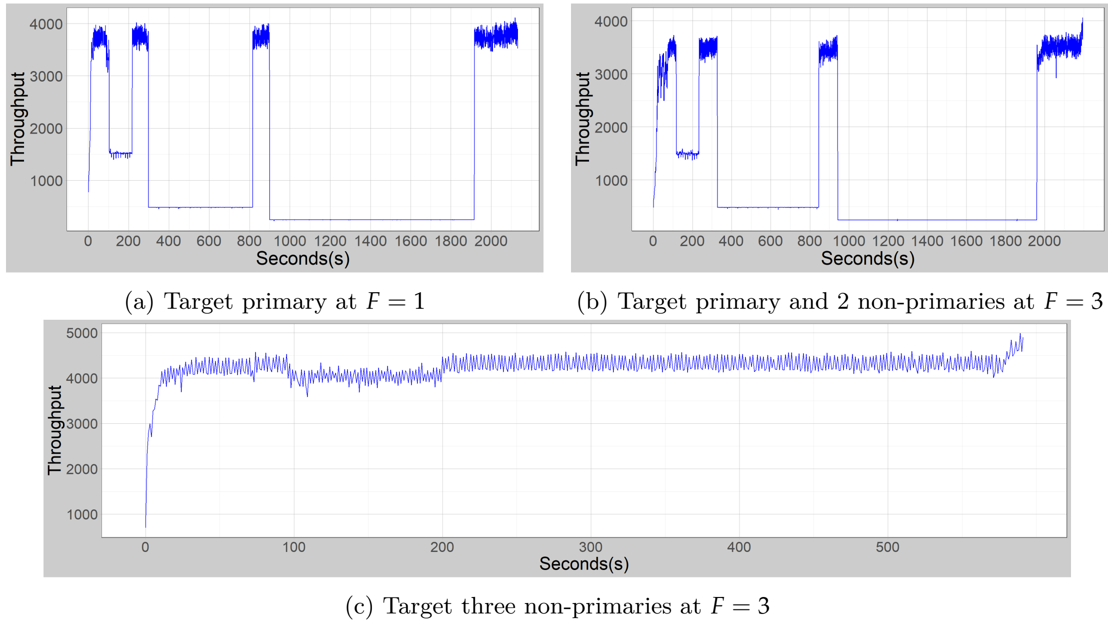
is indicative of the time non-primaries did not receive any ordered requests plus the view change process.

All results show similarity in the aspect that once the system is down to $2f + 1$, a drop in performance is followed along. The former statement also denotes that, between $3f + 1$ and $2f + 1$, the number of operations executed per second does not decrease, indicating that the system works as intended. This late decrease in performance can be attributed to either one or more of the correct replicas being slower than their peers or due to an internal change in the protocol to guarantee support for safety since it reached the minimum number of replicas for such.

5.3.2 Delaying Replicas

This experiment consisted of injecting and executing a fault schedule that delayed replicas (primaries and non-primaries) before sending messages independent of its type. Faults were triggered for 10000 consecutive consensus rounds, starting at consensus round 50000, 100000, and 150000, with delay intervals of 20 milliseconds, 50 milliseconds, and 100 milliseconds respectively.

Figure 5.3 illustrates the results obtained when targeting replicas in different configurations with this fault schedule: the primary (Figure 5.3a), the primary and two non-primaries (Figure 5.3b), and three non-primaries (Figure 5.3c).



(a) Target primary at $F = 1$

(b) Target primary and 2 non-primaries at $F = 3$

(c) Target three non-primaries at $F = 3$

Figure 5.3: Results from a delay fault schedule

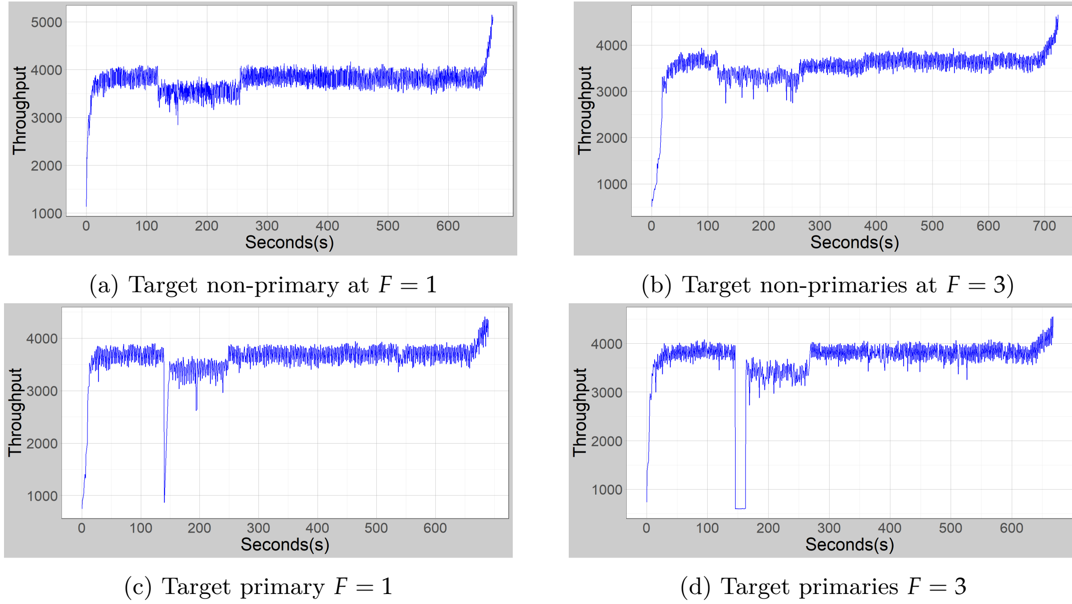
The outcomes observed in Figures 5.3a and 5.3b display that the systems were strongly affected by the faulty primary during the fault scheduling. It is to note that for $F = 3$, the collusion of faulty non-primaries and the primary did not yield a more substantial decline, as the system performance is directly related to the rate of PRE-PROPOSE messages sent by the primary. Lastly, we can recognize that the performance tends to drop even more due to increments in the delay that the primary takes to send messages.

In the third test, Figure 5.3c, we perceived that during the fault triggering done by the non-primaries, the performance deteriorated a little. This drop might be related to one or more correct replicas being a tad slower than their other correct peers.

5.3.3 Message dropper

This experiment consisted in injecting and executing a fault schedule that forced replicas (primaries and non-primaries) to drop messages bypassing message sending methods. Faults started triggering from the beginning of consensus round number 50000 and ran for a duration of 50000 rounds.

Figure 5.4 presents the results gathered when targeting: the non-primaries at a $F = 1$ (Figure 5.4a) and $F = 3$ (Figure 5.4b) configuration, and the non-primaries at a $F = 1$ (Figure 5.4c) and $F = 3$ (Figure 5.4d) configuration.



(a) Target non-primary at $F = 1$

(b) Target non-primaries at $F = 3$)

(c) Target primary $F = 1$

(d) Target primaries $F = 3$

Figure 5.4: Results for a message dropper schedule

According to the outcomes obtained for faulty non-primaries, Figures 5.4a and 5.4b, we can verify a slight decrease in the system's performance during the fault scheduling.

As for the results for faulty primaries, Figures 5.4c and 5.4d, a performance drop is seen whenever a primary stops transmitting messages, as this is referent to the period non-primaries did not receive any PRE-PROPOSE message from the primary plus the view change instance that occurred to replace that primary. We can also observe that for $F = 3$, after each successive view change, the former three primaries continued to trigger the fault as non-primaries, which can be seen in the performance drop period and correlated to the results from tests made with only faulty non-primaries.

In all elapsed tests, a slight performance drop is seen to happen during the fault scheduling. This reduction in performance is comparable to previous results where the number of non-faulty replicas reaches the minimum required for the system to guarantee safety.

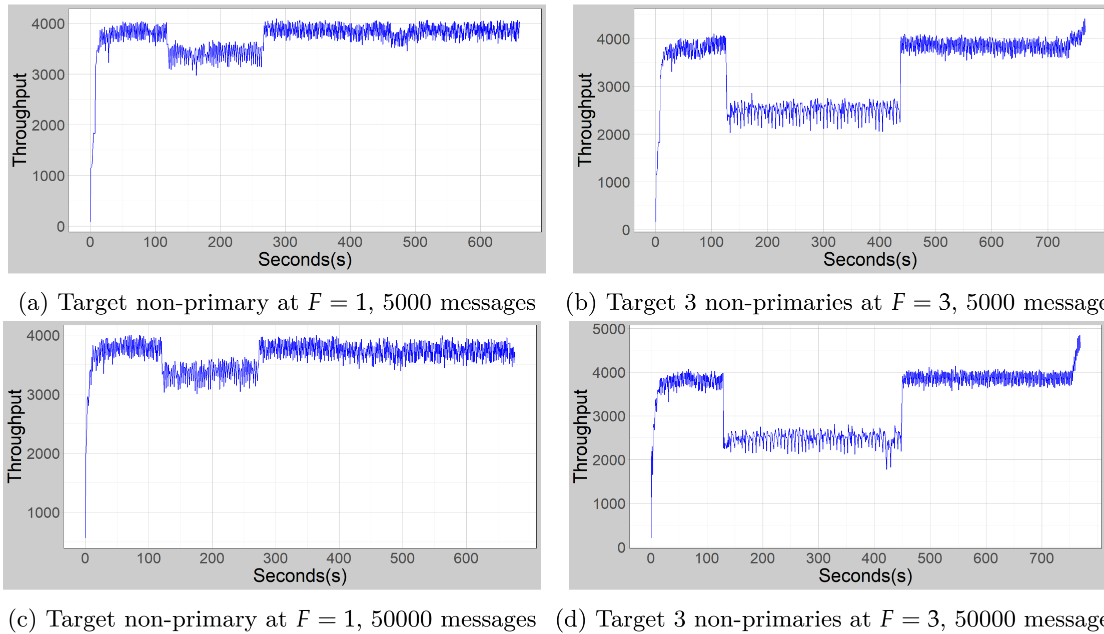
5.3.4 Flood attacks

This experiment consisted in injecting and executing a fault schedule that forces replicas to flood the system or a particular target with additional messages for each legitimate message sent. Faults were triggered at consensus round number 50000 and ran for 50000 rounds unless stated contrarily. The number of additional messages sent fluctuated between 5000 and 50000 messages.

We conducted three different tests for this experiment: a) a system flood produced by faulty non-primaries, b) a system flood carried by a faulty primary, and c) a targeted flood attack against the primary arranged by faulty non-primaries.

a) System Flood by non-primaries

We ran two different fault schedules for this experiment, where one relied on faulty non-primaries transmitting 5000 additional messages per valid message and another where they sent an additional 50000 messages per correct message.



(a) Target non-primary at $F = 1$, 5000 messages
(b) Target 3 non-primaries at $F = 3$, 5000 messages
(c) Target non-primary at $F = 1$, 50000 messages
(d) Target 3 non-primaries at $F = 3$, 50000 messages

Figure 5.5: Results from a flood fault schedule that targeted all system

Figure 5.5 presents the outcomes from this experiment where faulty non-primaries flood all replicas in the system. Figures 5.5a and 5.5b exhibit the results for $F = 1$ and $F = 3$, where 5000 additional messages were sent per legitimate message. As for figures 5.5c and 5.5d, these depict the results for $F = 1$ and $F = 3$ where 50000 additional messages were transmitted for each valid message.

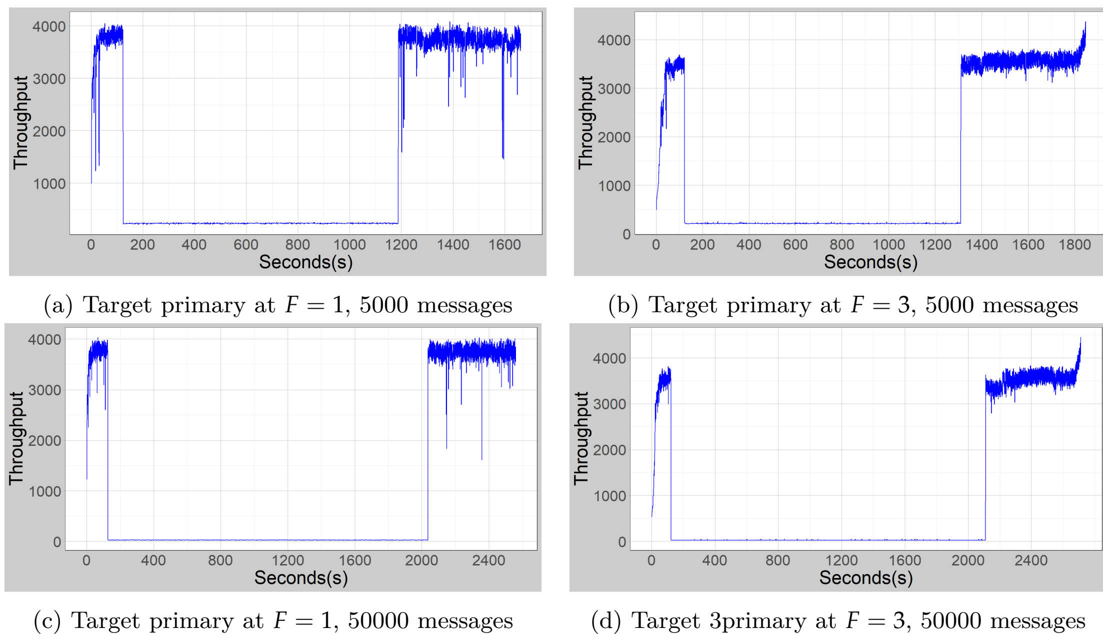
According to the outcomes above, a flood-based attack from faulty non-primaries against all correct replicas decreases the system's performance. It is to note that the attack does get stronger based on the number of replicas present on the consensus group. After all, the flood drove the system into a more extended deteriorated state for $F = 3$, while for $F = 1$, that period was shorter.

When comparing both the 5000 and 50000 experiments, we can see that there is barely any variation in the total number of operations executed per second during the execution

of the flood. As such, we can assume that BFT-SMaRt can mitigate these sorts of attempts at a certain threshold.

b) System flood by a faulty primary

We conducted two different fault schedules for this experiment, where one relied on a faulty primary transmitting 5000 additional messages and another where it sent an additional 50000 messages. However, due to time constraints (tests being quite long), we reduced the number of triggered rounds for the 5000 flood to 10000 rounds and 50000 flood to 2000 rounds.



(a) Target primary at $F = 1$, 5000 messages

(b) Target primary at $F = 3$, 5000 messages

(c) Target primary at $F = 1$, 50000 messages

(d) Target 3primary at $F = 3$, 50000 messages

Figure 5.6: Results from a flood fault schedule that targeted all system

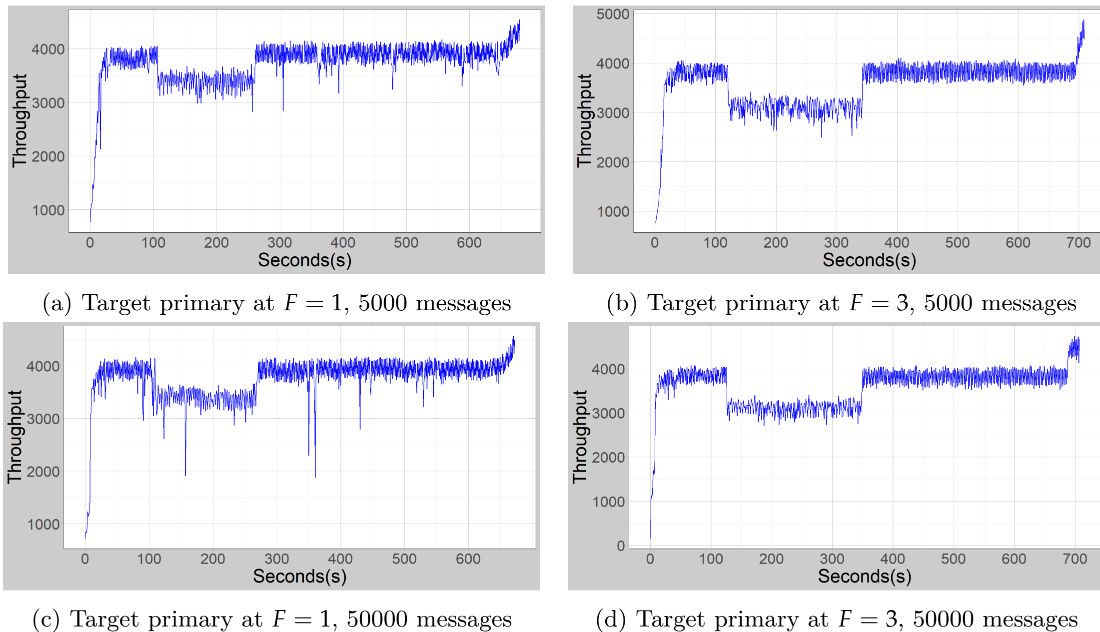
Figure 5.6 displays the results for this experiment where a faulty primary floods all non-primaries in the group. Figures 5.6a and 5.6b exhibit the results for $F = 1$ and $F = 3$ correspondingly, where 5000 additional messages were sent per valid message. Figures 5.6c and 5.6d depict the outcomes for $F = 1$ and $F = 3$ where 50000 additional messages were sent for each valid message.

Based on the gathered results, a flood attack conducted by a faulty primary causes many issues in the system's ability to process requests, leading it to a state where the system is almost inoperative. It is to note that the number of flood messages is sufficiently high to impact system performance without triggering a view change since new requests are still ordered before non-primaries associated timers are expired.

Lastly, when comparing both experiments based on the number of messages sent, we can see that increasing the strength of the flood does extend the degradation period the system experiences, especially since there is a considerable difference in the number of rounds that the flood was triggered for each experiment.

c) Targeted flood against primary conducted by faulty non-primaries

For this experiment, we conducted two similar fault schedules, where faulty non-primaries targeted the primary with a flood that ranged between the 5000 and 50000 additional messages per valid message sent. Faults were triggered at the start of the consensus round number 50000 and ran for 50000 rounds.



(a) Target primary at $F = 1$, 5000 messages

(b) Target primary at $F = 3$, 5000 messages

(c) Target primary at $F = 1$, 50000 messages

(d) Target primary at $F = 3$, 50000 messages

Figure 5.7: Results from a flood fault schedule that targeted all system

Figure 5.7 presents the results for this experiment where faulty non-primaries flood specifically the primary. Figures 5.7a and 5.7b display the results for $F = 1$ and $F = 3$ where 5000 additional messages were sent per legitimate message. Figures 5.7c and 5.7d depict the outcomes for $F = 1$ and $F = 3$ where 50000 additional messages were sent for each valid message.

The results here reveal that there is a degree of degradation applied to the system while a flood attack is targeting the primary of the system. This reduction in performance is essentially due to the primary wasting time reading and processing the additional messages it receives.

When thoroughly comparing the experiments here and those in the section where non-primaries flood the entire system, we see that flooding an entire system proves to be slightly more impactful than flooding only the primary.

5.4 Protocol specific faults

In this section, we tested the system against a more particular set of faults that target specific messages and functions, such as the agreement and view change phase and the usage of faulty clients to compromise the system. The experiments in this section were run for the $F = 3$ configuration since we only wanted to experiment with specific protocol elements. It is to note that the results discussed in this segment focus more on the ability to disrupt the system's state than the performance impact.

Faults run in the experiments in this section, with the exception of client-related tests, started at consensus round 50000 and were triggered for a variable number, which depended on the goal we wanted to achieve, i.e., initiate view changes deliberately.

Most arguments from the messages were modified, particularly the integer value ones, as they were easier to manipulate. It is worth noting that several different tests were generated for each argument based on the host (replica or client). Each argument had the following tests:

- randomized value;
- increments of 10 of the message's original argument value;
- decrements of 10 of the message's original argument value;
- faulty replicas send modified messages to only the primary, and the messages are altered based on one of the first three points in separate tests.
- faulty clients send messages with different argument values to different sub-sets of replicas, where the values are always the same for each respective replica;
- faulty clients send messages with different argument values to different sub-sets of non-primaries, where the values are always the same for each respective replica.

Additional tests related to flood-based attacks were also added for particular messages types, specifically view changes.

5.4.1 Agreement phase - REQUEST

REQUEST messages at a low level are coded with the following arguments: the client's id, the session id of the client, the sequence number created based on the message type, the operation sequence number disregarding message type, the command to be executed, the current view, and the type of the request. Most arguments, particularly those with an integer value, were modified and tested separately in different experiments as they are easier to manipulate.

For this experiment, we ran several different fault schedules applied to 10 faulty clients out of the total of 50, which involved modifying the contents of the REQUEST message to confuse the system into, for instance, initiate a view change. In addition, since faulty clients individually use their own request number as a metric for injection, the starting point chosen for fault triggering was the request number 10000. Faults were triggered for a variable amount of request transmission rounds, which depended on the experiment's objective.

The results of the experiments are summarized in the table 5.8.

	Client Session ID	Sequence Number	Operation ID	Current View	Experiment Result
Test 1	Injection Target	—	—	—	No Change
Test 2	—	Injection Target	—	—	No Change
Test 3	—	—	Injection Target	—	View Change Triggered
Test 4	—	—	—	Injection Target	No Change

Figure 5.8: Experimental results for REQUEST message modification

The summary (Figure 5.8) displays that most arguments that were modified did not present any change that could disrupt the system's state, with the exception of one, the "Operation ID".

In two of the many tests performed to the argument "Operation ID", view changes were triggered as a result. When changing this argument across different messages (where the argument is static for each different message related to different replicas) and subsequently sending them to different sub-sets of replicas or different sub-sets of non-primaries (primary receives no messages), view changes were consequently initiated. We believe that the view change protocol was initiated due to, non-primaries waiting to receive the ordered

requests they had in their ToOrder set but instead were not ordered by the primary despite retransmission attempts by each non-primary.

Figure 5.9 depicts the results for when faulty clients send messages with different operation ids (same ids for the same replica) to different replicas. This experiment was run for around 500 client rounds, starting at client round 10000.

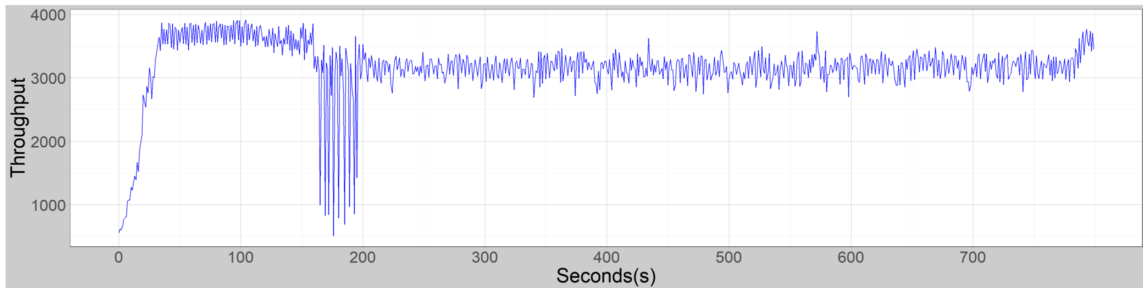


Figure 5.9: Results for $F = 3$ for when faulty clients send REQUESTS with different operation ids to different replicas.

The gathered results reveal multiple episodes of low throughput values across the test. Each of these descending values represents a view change that occurred midway through the test. After a couple of view changes, the protocol/replicas started detecting that it was a replay attack and consequently discarded REQUESTS that were similar afterward from those clients. The performance degradation observed after the last view change is due to faulty clients continuing to execute the fault setup. Since the REQUESTS from the faulty clients are never processed, the injection continues indefinitely because the round number for each faulty client never reaches the end trigger fault point. We can infer that it is possible to initiate view changes by using faulty clients deliberately.

Using this setup, we expanded the experiment to a collusion between faulty clients and faulty non-primaries. This experiment revolved around making the faulty non-primary into the primary through induced view changes from the attacks from faulty clients. Once it became the primary, it would delay message transmission by 100 milliseconds every time it wanted to send a message. The experiment was run for 500 client rounds, starting at client round 10000.

The results presented in figure 5.10 show that the system can suffer from a colluding attack between faulty clients and faulty non-primaries. Once the faulty non-primary became the primary, the entire system's performance went down because of the injected delay. It is to note that we stopped the test midway, only to show that this situation is existent.

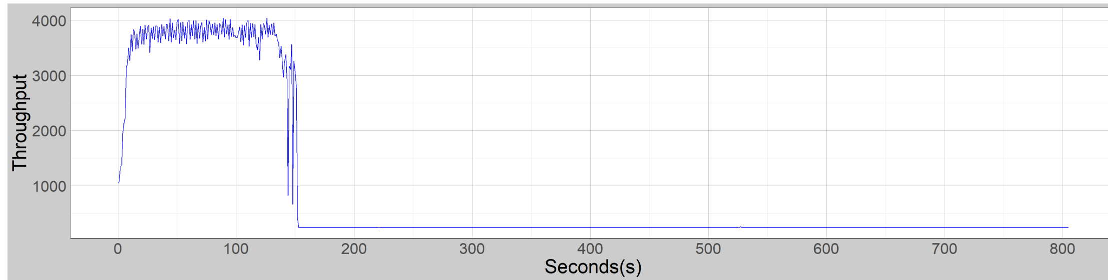


Figure 5.10: Results for $F = 3$ for when faulty clients collude with a faulty-primary to delay the system.

5.4.2 Agreement phase - PROPOSE

PROPOSE messages consist of the consensus id (sequence number), the epoch (view number), the replica id and a value (request to be executed). We focused our efforts solely in two arguments, the sequence and view number as they were easy to manipulate.

For this experiment, different fault schedules were run where the primary was the sole faulty replica, which involved modifying the contents of the PROPOSE message. Faults were triggered at the round of consensus 50000 for 50000 consensus rounds. Figure 5.11 presents the summary of tests performed under these circumstances.

	Consensus ID	View Number	Experiment Result
Test 1	Injection Target	———	View Change Triggered
Test 2	———	Injection Target	View Change Triggered

Figure 5.11: Experimental results for PROPOSE message modification.

As expected, throughout the experiments, the faulty primary was replaced through a view change whenever it tried to send any altered message.

5.4.3 Agreement phase - WRITE

Like the PROPOSE message, WRITE message follows a similar pattern which consist of the consensus id (sequence number), the epoch (view number), the replica id and a value (request to be executed). Similarly as the previous experiment, only two arguments were manipulated, the sequence and view number. Faults were triggered at the start of consensus round 50000 for 50000 rounds in total. The following Figure 5.12 displays the results of the experiments.

	Consensus ID	View Number	Experiment Result
Test 1	Injection Target	——	No Change
Test 2	——	Injection Target	No Change

Figure 5.12: Experimental results for WRITE message modification.

The findings revealed no evidence of a disruption in the system's state or a significant impact on performance. These attacks were ineffective because at least $2f+1$ correct replicas contributed to consensus. The perceived performance impact is comparable to the results of when faulty replicas did not contribute to consensus, such as message dropping from faulty non-primaries or faulty non-primaries crashing.

5.4.4 Agreement phase - ACCEPT

As in the previous phases of the agreement, ACCEPT messages use a similar pattern consisting of the consensus id (sequence number), the epoch (view number), the replica id, and a value (request to be executed). Only two arguments were modified, the sequence and view number. Faults were triggered at the start of consensus round 50000 for 50000 rounds in total. Figure 5.13 presents the results of the elaborated tests.

	Consensus ID	View Number	Experiment Result
Test 1	Injection Target	——	No Change
Test 2	——	Injection Target	No Change

Figure 5.13: Experimental results for WRITE message modification.

The findings revealed no evidence of a disruption in the system's state or a significant impact on performance. These attacks were ineffective because there were at least $2f+1$ correct replicas contributing to consensus. The perceived performance impact is comparable to the results of when faulty replicas did not contribute to consensus, such as message dropping or replica crashing.

5.4.5 View change phase - STOP

STOP messages consist of the next view number and a set of requests that were time-outed. We only modified the view number as it was easy to manipulate. In addition, we tested with flood attacks, where faulty replicas only transmitted STOP messages during its fault schedule. Faults were triggered at the start of consensus round 50000 for 50000 rounds in total. Figure 5.14 presents the results of the elaborated tests.

	View Number	Experiment Result
Test 1	Injection Target	No Change
Test 2 (Flood)	—	View Change Triggered; Non-faulty Replicas Crash; System Invariance Broken;

Figure 5.14: Experimental results for STOP message modification.

Throughout the tests, we observed that nothing would happen when modifying the values for the view number, which was to expect as there was at least the minimum number of correct replicas in execution in the group.

In the second experiment, we found out that flood attacks with this message type created situations where correct replicas would crash due to the depletion of memory or storage. This situation evolved into one where all correct replicas were crashing. It is to note that, once the first correct replica crashed, the invariance of the system broke, and as such, the group could not progress any further.

Because we know that correct replicas can crash due to resource depletion caused by a flood of STOP messages, we expanded the test to one where faulty non-primaries flood only the primary in order to provoke a view change.

For this experiment, two faulty non-primaries focused their attacks only on the primary of the system. The reason we use two faulty non-primaries is because if we use three faulty replicas, we risk breaking the system's invariance once the primary crashes.

The flood attack started triggering at the round of consensus 50000 and ran for 50000 rounds. The following Figure 5.15 displays the results of the experiment.

The results exhibited in Figure 5.15 testify that it is possible to compromise a correct primary replica via a flood of STOPS anytime we want. We see that right after the flood

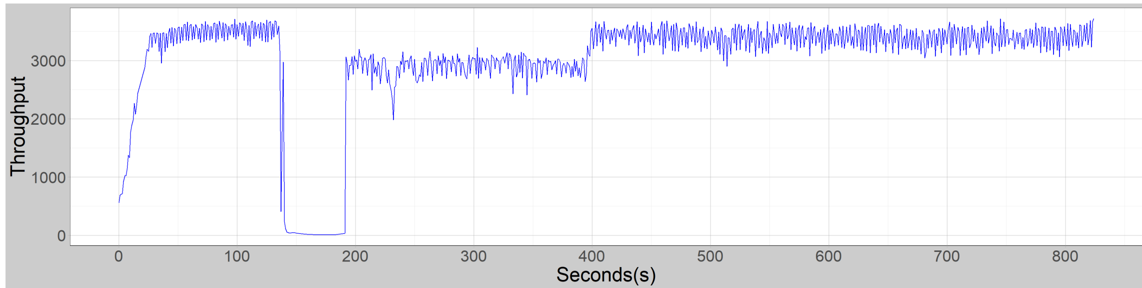


Figure 5.15: Results for $F = 3$ for when a correct primary gets replaced via a flood of STOP messages.

is injected against the correct primary, it tends to decrease its performance tremendously, which is most likely due to a tentative to process all incoming STOP messages. During this period, the primary could still order some requests but at a sluggish pace. Once the primary reached a resource depletion state and subsequently crashing, a view change happened, and a new primary was elected.

From around the 190-second mark till the 400-second mark, we see that the system is under the effect of a significant performance impact. This is caused by the two faulty non-primaries still attacking the former primary (which crashed) and because the former primary was not functional. This situation is comparable to many tests in the section 5.3, considering that once the system reached the $2f + 1$ scenario, a performance impact would occur. It is worth noting that

It is worth noting that we can further amplify this situation into one where we deliberately initiate a view change through a targeted flood to replace a correct primary into one of the faulty non-primaries.

5.4.6 View change phase - STOPDATA

STOPDATA messages consist of the next view number and their decision logs. Only the view number was modified. Like the previous phase of this protocol, it was also tested with flood attacks while only using STOPDATA messages. Faults were triggered at the start of the consensus round 50000 for 50000 in total. Figure 5.16 displays the summary of the experiment results.

The outcomes from this experiment revealed very similar results to the previous phase of the view change. Nothing happened whenever we modified the view number, as the system still had the correct minimum of replicas to continue processing.

	View Number	Experiment Result
Test 1	Injection Target	No Change
Test 2 (Flood)	—	View Change Triggered; Non-faulty Replicas Crash; System Invariance Broken;

Figure 5.16: Experimental results for STOPDATA message modification.

The second test exposed comparable results as the STOP phase, where a STOPDATA flood attack would lead correct replicas into crashing due to their memory being depleted. Furthermore, we also had situations where the invariance of the system broke, which happened whenever the number of correct replicas was lower than the minimum threshold for consensus/processing to progress. Lastly, similarly to the STOP phase, it was possible to initiate view changes deliberately with a targeted flood attack against the system's primary.

5.4.7 View change phase - SYNC

SYNC messages consist of the view number and a proof (agreed upon consensus instances). It is to note that only the primary was the sole faulty replica. Only the view number was altered. In this phase, we also flooded the system with SYNC messages. Faults were triggered at the start of the consensus round 50000 for 50000 in total. The following Figure 5.17 presents the results of the tests.

	View Number	Experiment Result
Test 1	Injection Target	View Change Triggered
Test 2 (Flood)	—	View Change Triggered

Figure 5.17: Experimental results for SYNC message modification.

The results showed that whenever the faulty primary tried to send any message with a modified view number, a view change would be triggered to replace it. As for the

flood attack, the faulty primary ended up delaying the system by mere seconds, till it was replaced later on, which was expected.

Chapter 6

Conclusion and Future Work

In this document, we discussed the model basis of how byzantine fault-tolerant protocols work, as we also reviewed several protocols, specifically deterministic, in order to find proven and possible theoretical weaknesses that a fault injector could later explore. We found that deterministic protocols, especially those that use a primary, have various issues that could lead to several types of attacks being used to exploit them.

As for the second point, we proposed a new fault injector, Zermia, capable of maintaining its level of intrusiveness low due to the use of aspect-based functions to inject faults. This novel fault injector is also the first, to this date, to employ gRPC to trade information between a Coordinator server and an Agent to reduce the level of communication complexity. Additionally, it is shown that it can inject faults in multiple replicas and clients and can employ a simple priority system to maximize the potential damage to a system/replica.

The experimental evaluation of BFT-SMaRt revealed that we were successful in injecting diverse faults into it, as the results of some tests revealed interesting behaviors and performances that had not yet been verified. These tests also revealed that BFT-SMart, as expected, is prone to faults that involve the system's primary. It also revealed new attacks possible to be made against BFT-SMaRt, such as crashing correct replicas in the protocol with flood-based attacks, which depleted the memory of the replicas. We also were capable of using faulty clients to disrupt the protocol into the view change phase whenever we wanted.

As for the application's future, it could be beneficial to use this injector to assess other distributed systems aside from BFT protocols, given that a multitude of these faults can affect them, but the focus, for now, is purely on testing BFT protocols. One of the biggest hurdles will be deploying this application on other protocols since it will be needed to

understand their underline structure to adapt it. As such, it will be required to restructure the application's architecture into one that has a better capability to do it so. As for now, Zermia only works with BFT-SMaRt, and the variety of attacks has yet to be further expanded to include faults such as MAC/signature forging attacks.

Bibliography

[1] R. A. Quinnell, "Distributed operating systems combine multiple processors into a single machine." EDN, vol. 40, no. 20, pp. 38–43, 1995. [Cited on page 2.]

[2] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 45–48, New York 2007. [Cited on pages xvii, 2, 3, 12, 17, 22, 23, 26, and 50.]

[3] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," Proc. NSDI, pp. 169–184, 2009.

[4] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 363–376. [Cited on page 2.]

[5] A. Shoker, J.-P. Bahsoun, , and M. Yabandeh, "Improving independence of failures in bft," Network Computing and Applications (NCA), 2013 12th IEEE International Symposium, pp. 227–234, Aug 2013. [Cited on page 2.]

[6] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-scalable byzantine fault tolerant services," SOSP, 2005. [Cited on page 2.]

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in Proceedings of the 7th symposium on Operating systems design and implementation, 2006, pp. 177–190. [Cited on pages 2 and 17.]

[8] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," IEEE Transactions on Computers, vol. 62, no. 1, pp. 16–30, 2011. [Cited on page 2.]

[9] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," ACM Transactions on Computer Systems (TOCS), vol. 20, no. 4, pp. 398–461, 2002. [Cited on pages xvii, 2, 12, 13, 17, 18, 23, 27, 36, 45, and 47.]

[10] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2014, pp. 355–362. [Cited on pages 2, 4, 17, 27, 42, 45, 55, 71, and 79.]

[11] J.-P. Bahsoun, R. Guerraoui, and A. Shoker, "Making bft protocols really adaptive," in 2015 IEEE International Parallel and Distributed Processing Symposium. IEEE, 2015, pp. 904–913. [Cited on page 2.]

[12] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," NSDI, vol. 9, pp. 153–168, 2009. [Cited on pages 2, 17, and 37.]

[13] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in 2009 28th IEEE International Symposium on Reliable Distributed Systems. IEEE, 2009, pp. 135–144. [Cited on page 12.]

[14] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN). IEEE, 2008, pp. 197–206. [Cited on pages xvii, 12, 17, 32, 33, 35, 37, 49, and 58.]

[15] P.-L. Aublin, S. B. Mokhtar, and V. Qu´ema, "Rbft: Redundant byzantine fault tolerance," IEEE 33rd International Conference on Distributed Computing Systems, pp. 297–306, July 2013. [Cited on pages xvii, 2, 37, 38, 39, 60, and 62.]

[16] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 31–42. [Cited on page 2.]

[17] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 2028–2041.

[18] F. Shen, Y. Long, Z. Liu, H. Liu, D. Gu, and N. Liu, "A practical dynamic enhanced bft protocol," International Conference on Network and System Security, NSS, pp. 288–304, 2019.

[19] C. Liu, S. Duan, and H. Zhang, "Epic: Efficient asynchronous bft with adaptive security," 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 337–451, 2020. [Cited on page 2.]

[20] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo, "Experiences with fault-injection in a byzantine fault-tolerant protocol," Proc. of ACM/IFIP/ USENIX Middleware'13, 2013. [Cited on pages 3 and 42.]

[21] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for bft testing," arXiv preprint arXiv:2004.10617, 2020. [Cited on pages 3, 42, and 43.]

[22] D. Gupta, L. Perronne, and S. Bouchenak, "Bft-bench: Towards a practical evaluation of robustness and effectiveness of bft protocols," in Distributed Applications and Interoperable Systems, 2016, pp. 115–128. [Cited on pages 3, 42, and 62.]

[23] grpc : A high-performance, open source universal rpc framework. [Online]. Available: https://github.com/grpc [Cited on pages 4 and 71.]

[24] J. C. de Sousa, "Byzantine state machine replication for the masses," Ph.D. dissertation, Universidade de Lisboa (Portugal), 2017. [Cited on pages xvii, 4, 27, 28, 30, 42, 45, 55, and 71.]

[25] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," ACM Computing Surveys (CSUR), vol. 22, no. 4, pp. 299–319, 1990. [Cited on page 7.]

[26] C. Cachin, "Security and fault-tolerance in distributed systems," IBM Zurich Research Lab, 2006. [Cited on pages 9 and 10.]

[27] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," Journal of the Association for Computing Machinery, 1985. [Cited on page 10.]

[28] M. Correia, G. S. Veronese, N. F. Neves, and P. Verissimo, "Byzantine consensus in asynchronous message-passing systems: a survey," International Journal of Critical Computer-Based Systems, vol. 2, no. 2, pp. 141–161, 2011. [Cited on page 11.]

[29] G. Bracha, "Asynchronous byzantine agreement protocols," Information and Computation, pp. 130–143, 1987. [Cited on page 11.]

[30] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," ACM TOPLAS, pp. 382–401, 1982. [Cited on pages xvii, 13, 14, 15, 16, and 17.]

[31] M. Castro, B. Liskov et al., "Practical byzantine fault tolerance," in OSDI, vol. 99, no. 1999, 1999, pp. 173–186. [Cited on pages 17, 23, 27, 45, and 47.]

[32] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché, "Upright cluster services," Proc. of the ACM SOSP'09, 2009. [Cited on page 17.]

[33] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," Proceedings of the 2013 USENIX Annual Technical Conference, San Jose, CA, USA , 2013. [Cited on pages xvii and 31.]

[34] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "Modifi: a model-implemented fault injection tool," in International Conference on Computer Safety, Reliability, and Security. Springer, 2010, pp. 210–222. [Cited on page 40.]

[35] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," Computer, vol. 30, no. 4, pp. 75–82, 1997.

[36] N. Song, J. Qin, X. Pan, and Y. Deng, "Fault injection methodology and tools," in Proceedings of 2011 International Conference on Electronics and Optoelectronics, vol. 1. IEEE, 2011, pp. V1–47. [Cited on page 40.]

[37] H. Ziade, R. A. Ayoubi, R. Velazco et al., "A survey on fault injection techniques," Int. Arab J. Inf. Technol., vol. 1, no. 2, pp. 171–186, 2004. [Cited on pages 40 and 41.]

[38] R. Kumar, P. Jovanovic, and I. Polian, "Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis," 2014 IEEE 20th International On-Line Testing Symposium (IOLTS), pp. 43–48, 2014. [Cited on page 41.]

[39] M. Madau, M. Agoyan, J. Balasch, M. Grujić, P. Haddad, P. Maurine, V. Rožić, D. Singelée, B. Yang, and I. Verbauwhede, "The impact of pulsed electromagnetic

fault injection on true random number generators," 2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 43–48, 2018. [Cited on page 41.]

[40] M. Dumont, M. Lisart, and P. Maurine, "Electromagnetic fault injection : How faults occur," 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 9–16, 2019.

[41] A. Cui and R. Housley, "Badfet : Defeating modern secure boot using second-order pulsed electromagnetic fault injection," 11th USENIX Workshop on Offensive Technologies (WOOT 17), 2017. [Cited on page 41.]

[42] P. Yuste, D. Andrès, L. Lemus, J. Martin, and P. Gil-Vicente, "Inerte: Integrated nexus-based real-time fault injection tool for embedded systems," International Conference on Dependable Systems and Networks, 2003. [Cited on page 41.]

[43] J. Arlat, Y. Crouzet, and J. C. Laprie, "Fault injection for dependability validation of fault-tolerant computer systems," Proc. 19th International Symp. on Fault-Tolerant Computing, pp. 348–355, 1989. [Cited on page 41.]

[44] H. Madeira, M.Rela, F.Moreira, and J.Silva, "Rifle: A general purpose pin-level fault injector," Proc. First European Dependable Computing Conference, pp. 199–216, 1994. [Cited on page 41.]

[45] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," IEEE micro, pp. 8–23, 1994. [Cited on page 41.]

[46] J. Carreira, H. Madeira, J. G. Silva et al., "Xception: Software fault injection and monitoring in processor functional units," Dependable Computing and Fault Tolerant Systems, vol. 10, pp. 245–266, 1998. [Cited on page 41.]

[47] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," IEEE Transactions on computers vol. 44, pp. 248–260, 1995.

[48] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders, "A global-state-triggered fault injector for distributed system evaluation," IEEE Transactions on Parallel and Distributed Systems vol. 15, pp. 593–605, 2004.

[49] S. Han, K. G. Shin, and H. A. Rosenberg, “Doctor: An integrated software fault injection environment for distributed real-time systems,” Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium, pp. 204–213, 1995. [Cited on page 41.]

[50] P. Sousa, N. F. Neves, and P. Verissimo, “Hidden problems of asynchronous proactive recovery,” in Proceedings of the Workshop on Hot Topics in System Dependability, 2007. [Cited on page 50.]

[51] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin, “Revisiting fast practical byzantine fault tolerance,” ArXiv, Dec 2017. [Cited on page 51.]

[52] I. Abraham, G. Gueta, D. Malkhi, and J. Martin, “Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma,” ArXiv, Jan 2018. [Cited on page 54.]

[53] L. Perronne and S. Bouchenak, “Towards efficient and robust bft protocols,” in Fast Abstract in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2016. [Cited on page 58.]

[54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Ycsb : Yahoo! cloud serving benchmark. [Online]. Available: https://github.com/brianfrankcooper/YCSB [Cited on page 79.]

[55] Cooper, Brian F and Silberstein, Adam and Tam, Erwin and Ramakrishnan, Raghu and Sears, Russell, “Benchmarking cloud serving systems with ycsb,” in Proceedings of the 1st ACM symposium on Cloud computing, 2010, pp. 143–154. [Cited on page 79.]