Individual Diploma Thesis

IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF BYZANTINE-TOLERANT DISTRIBUTED SETS

Loukas Papalazarou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2023

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

Implementation and Experimental Evaluation of Byzantine-Tolerant Distributed Sets

Loukas Papalazarou

Supervisor: Chryssis Georgiou

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining a degree in Computer Science from the Department of Computer Science of the University of Cyprus

May 2023

Acknowledgments

I would like to express my heartfelt gratitude to Professor Chryssis Georgiou, my esteemed supervisor, for providing me with the invaluable opportunity to work on this fascinating project. His consistent support and expert guidance have been instrumental in navigating the complexities of this research and producing the best results possible. Additionally, I would like to extend my heartfelt appreciation to my family for their unwavering support throughout this academic journey. Their constant encouragement and belief in me have been instrumental in making this achievement possible. Lastly, I would like to express my deep gratitude to the Computer Science Department of the University of Cyprus for equipping me with invaluable knowledge and skills that have empowered me to overcome any challenge.

Abstract

Recently, there has been a significant increase in the use of distributed systems in various industries. The demand for greater scalability, dependability, and availability in applications like cloud computing, big data, and the Internet of Things has fueled the growth of distributed systems. Also, the emergence of distributed systems has been aided by the rising acceptance of decentralized systems like blockchain technology. Naturally, any distributed system will be required to operate on numerous machines concurrently, thus introducing several new risks and threads that the traditional client-server architecture did not have to face.

This thesis presents a novel contribution to the field of distributed systems by proposing, implementing, and experimentally evaluating a Byzantine Fault Tolerant Distributed G-Set system using the Go Programming Language and the ZeroMQ Messaging Library. The G-Set is a data type which maintains records, that may only grow (grow-only sets, as deleting of a record is not allowed). Each participating server holds a local copy of a set of records which remains consistent among other participants' sets after the addition of any record to the G-Set. Unlike traditional ledgers, the G-Set does not require the records to be in any specific order. This system will be commonly referred to as BDSO, which stands for *"Byzantine fault-tolerant Distributed Set Object"*. The proposed BDSO ensures correctness in the presence of Byzantine faults, up to a threshold of (n-1)/3 malicious participants, where n represents the total number of participants. This system is then used to address a variation of the 2-Atomic-Appends problem, the 2-Atomic-Adds problem, by developing a special type of BDSO called Smart BDSO (SBDSO). The SBDSO is used as an intermediary between 2 other BDSOs, thus controlling the additions of records to any of the BDSOs while adhering to the atomic criteria. Such strong guarantees come at a cost, which in this case is the large number of messages exchanged within the system.

The experimental evaluation of these systems was conducted on a fully distributed network of machines over the CloudLab testbed. It covered various aspects, such as operation latency and message complexity, to gather as much information as possible about the algorithms' scalability and suitability for commercial use. The evaluation aimed to assess the systems' performance in a real-world setting and provide valuable insights into their behavior under different network conditions and attack scenarios.

Contents

Chapter 1

Introduction

1.1 Motivation

Distributed ledger technology [28], commonly known as "blockchain", has gained significant attention in recent years due to its potential to revolutionize various industries by providing a secure and decentralized way to store and transfer data. However, the decentralized nature of blockchain systems also introduces new challenges, particularly when it comes to fault tolerance.

Blockchain systems rely on consensus mechanisms to ensure that all nodes in the network agree on the state of the ledger. However, in a decentralized system, some nodes can fail or behave maliciously, leading to inconsistencies in the ledger and thus compromising the integrity of the system. This is known as the Byzantine Fault Tolerance (BFT) problem [9]. Various BFT algorithms have been proposed to address this problem, but many of them make assumptions about the synchrony of the system that may not be held in practice. Moreover, most of these algorithms have not been extensively evaluated in a blockchain setting.

Therefore, there is a need to develop and evaluate BFT algorithms specifically designed for blockchain systems that can handle the asynchronous nature of these systems. This thesis aims to contribute to this field by implementing a proposed Byzantine-tolerant distributed

ledger-like system [4] and evaluating its performance and scalability in a real-world setting. Unlike the traditional ledger system, our distributed system will host a *set* of records, we call the G-Set. The order of records within the G-Set is inconsequential, however, the content itself holds significance. The primary objective is to ensure that all servers possess an identical copy of the G-Set, irrespective of the individual order of records. This system will act as the backbone of another system we will be developing to solve the 2-Atomic-Adds problem. This is motivated by the growing importance of blockchain technology and the need for robust fault tolerance mechanisms to ensure the stability and integrity of these systems. Further elaboration on the 2-Atomic-Adds problem will be provided in subsequent chapters of this thesis.

1.2 Objective and Contribution

The main objective of this thesis is to implement and evaluate the algorithm in [4] for a Byzantine-Fault tolerant distributed ledger-like object, called the G-Set. G-Set is a grow-only set of records that supports two operations. The *get* operation which retrieves all the records, and the *add* operation which adds a new record to the set. By testing the algorithm in some real-world scenarios, we are looking to validate its correctness, evaluate its performance and its applicability. Consensus algorithms are commonly utilized in distributed ledger systems as the dominant approach. In contrast, this thesis introduces an innovative methodology for constructing distributed systems by employing Reliable Broadcast Algorithms and Sets in lieu of ledgers. By adopting this novel approach, the research demonstrates a prospective solution to the 2-Atomic-Adds Problem, a renowned and intricate challenge within the realm of distributed computing. Briefly, this problem involves the need to add records into 2 different G-Sets atomically, meaning either both will be added, or none. In addition to introducing a new kind of methodology, this thesis distinguishes itself by being the first-ever to provide comprehensive high-level algorithms, along with their implementation and experimental evaluation. It goes beyond theoretical exploration and offers practical insights into the proposed approach, making it a significant contribution to the field.

1.3 Methodology

We wanted to approach this thesis modularly, studying each part of the theoretical and practical parts separately, allowing us to be competent in all aspects of the thesis. Our first task was to understand the fundamental data structures and algorithms behind Distributed Ledger systems, Initially, we studied some previous work on the formalization of Distributed Ledger Objects [1]. We then explored a basic problem when dealing with multiple DLOs, that of the Atomic Appends [2]. Consequently, we gained a satisfactory understanding of the conditions in which a system of multiple DLOs requires a Smart DLO (SDLO) to act as a moderator between the involved DLOs. The next step was to consider asynchrony and Byzantine Faults to the atomic appends problem. The work on [3] neatly solves this problem and presents an experimental evaluation of the proposed algorithm, critical for our work. Having gained all the prior knowledge required to understand the problem at hand, we proceeded in studying [4] which generalizes the problem of Byzantine Fault Tolerance in distributed systems and solves it by introducing a new concept, the Grow-only set (G-Set). For the codebase, we took inspiration from a similar thesis by Vasilis Petrou [13].

Naturally, as we now understood the problem well, we started familiarizing ourselves with the programmatic tools we aimed to use. That being the Go programming language, along with the ZeroMQ messaging library, specifically its Dealer-Router messaging pattern, using [18, 21, 22].

With significant background knowledge of the theoretical parts and decent familiarity with the programmatic tools, we were now able to start the implementation. We began by implementing Bracha's Reliable Broadcast (BRB) algorithm [10], to confirm that it works on its own since the rest of the code relies on the correctness of BRB. After testing and confirming that BRB works, we implemented an entire local system with servers and clients represented by threads. This gave us an idea of how the system would function with numerous agents. Having the base of the system working properly, we moved on to the "remote" implementation. With the help of some automation scripts, we were able to configure remote machines to run our code. Then, the change of the clients and servers from threads to real machines meant we had a fully functioning system. We called this system "BDSO", of which the meaning will be explained in subsequent chapters.

As the BDSO system was now complete, we moved on to developing an extension of this system, what we called “2-Atomic-Adds”. This system was our solution to the 2-Atomic-Adds problem. The main idea was to take a BDSO server network and make its servers “Smart”, meaning that they could now understand atomic messages and how to handle them. After figuring out the topology and connecting the various server networks we had our final solution to the 2-Atomic-Adds problem.

With the desired systems in place, it was time to create some malicious actors and automated clients for testing. We planned out our tests, configured numerous machines, created automation scripts, and proceeded to run our experiments, both on the standalone BDSO system, as well as the 2-Atomic-Adds system using the CloudLab [23] testbed. Finally, we extracted the results and analyzed them.

In general, the implementation of the system was done using an Agile approach, meaning we were coding and solving problems on the fly, with only a rough idea of how the final codebase would look like.

1.3 Document Structure

Chapter 2 provides a comprehensive overview of the essential background knowledge required to grasp the topics discussed in this thesis. It delves into the fundamental principles of Distributed Computing, explores the key concepts related to Byzantine Faults, and covers the implementation tools, namely Golang and ZeroMQ. In Chapter 3, we provide the fundamental algorithms that this thesis is based on, and in Chapter 4 we provide the implementation details of a system we have built upon these algorithms. Subsequently, in Chapter 5, we evaluate our system under various Byzantine scenarios, and finally, in Chapter 6 we discuss our conclusions, findings as well as the difficulties we faced. The visual aids presented throughout this thesis were crafted using Canva [26] and Diagrams.net [27].

Chapter 2

Background

2.1 The Go Programming Language

To implement the programmatic aspect of this thesis, we opted to use the versatile and efficient programming language, Golang [15].

2.1.1 Introduction and Features

Go, also known as Golang, is a modern programming language developed by Google in 2007. It was designed to address some of the shortcomings of existing languages and improve large-scale software development's speed and efficiency. Go is a statically typed, compiled language. Its authors, Robert Griesemer, Rob Pike, and Ken Thompson wanted to create a programming language that would combine the speed of C with the syntactic simplicity of Python and include features such as garbage collection and concurrency.

Go's built-in concurrency support is one of its main selling points. It provides the goroutines functionality, which is its implementation of multithreaded-like behavior, and channels for communication between them. Go's concurrency model is based on Communicating Sequential Processes (CSP) which eliminates the need to consider the order of message delivery and it has been widely used in the field of concurrent programming and distributed systems.

Go's goroutines, garbage collector, and efficient memory management make it an excellent choice for large-scale software development. The standard library in Go provides a wide range of functionalities, including support for networking, encryption, and web development. Go also supports cross-compilation, meaning it can be compiled and run on multiple operating systems.

2.1.2 Why Go

The reason we chose to implement this project using Go is its server-oriented design and robust features provided. As mentioned before, Go was designed to overcome the shortcomings of other popular programming languages. Although our tasks were not computationally demanding, we had some network overheads which meant that optimizing every other aspect of the code in terms of the execution time was crucial. Pairing that with the relatively short time that the code had to be completed, meant that we needed an easy-to-understand, fast, and feature-rich programming language.

Although Golang has amazing concurrency support, we chose to only use the goroutines part and opted for an alternative way of communication instead of channels. The algorithm was implemented in both a local scenario and a LAN scenario and Golang's channels are not able to communicate with remote processes. Therefore, we used the ZeroMQ messaging library for both local and remote scenarios.

2.2 The ZeroMQ Messaging Library

To enable seamless communication among the various nodes of our system, we leveraged the powerful and flexible messaging library, ZeroMQ [20].

2.2.1 Introduction and Features

ZeroMQ (aka ØMQ, 0MQ, or zmq) is a lightweight messaging library designed by iMatrix aimed at use in distributed or concurrent applications. As described on the ZeroMQ website, this library "looks like an embeddable networking library but acts like a concurrency framework". The Zero in ZeroMQ is meant to symbolize the zero broker, meaning there is no middleman between the sender and receiver organizing the messaging queues, but instead, this happens from point to point.

2.2.2 ZeroMQ Strings

By default, ZeroMQ uses lists of strings to represent its messages. This is because lists of strings are a flexible and universal way to represent and transmit data, they are easy to integrate with other systems and languages, easy to manipulate, and most importantly, they are human-readable which makes debugging and troubleshooting way easier. The way we use handle messages as lists of strings will become apparent later with some code snippets.

2.2.3 ZeroMQ Sockets

ZeroMQ by default provides numerous socket types and messaging patterns derived from combinations of said sockets. The type of socket dictates its behavior.

Conventional sockets, in general, provide a synchronous interface to either connection-oriented reliable byte streams (SOCK STREAM) or connection-less unreliable datagrams (SOCK DGRAM). ZeroMQ sockets, in contrast, offer an abstraction of an asynchronous message queue, with the precise queueing semantics varying based on the socket type being used. ZeroMQ sockets deliver discrete messages, as opposed to traditional sockets that transfer discrete datagrams or streams of bytes.

2.2.4 Messaging Patterns and Dealer–Router

As mentioned earlier, ZeroMQ supports several messaging patterns, the most important being the following:

- Request-Reply: This pattern allows a client to send a request to a server and wait for a response. It's a one-to-one communication pattern, which makes it suitable for synchronous request-response scenarios.
- Publish-Subscribe: This pattern allows a publisher to send messages to multiple subscribers, who can receive the messages in parallel. It's a one-to-many communication pattern, which makes it suitable for asynchronous event-driven scenarios.
- Pipeline: This pattern allows multiple nodes to work together in a pipeline, where each node receives messages from the previous node, processes them, and then sends the results to the next node. It's a many-to-many communication pattern, which makes it suitable for parallel processing scenarios.
- Exclusive Pair: This pattern allows two nodes to communicate with each other in a one-to-one fashion and it's similar to the request-response pattern, but it is not limited to client-server communication.

And finally, there is **Dealer – Router**, which is the one we have used. The Dealer-Router pattern is another messaging pattern provided by ZeroMQ. It allows a set of worker nodes (Dealers) to send requests to a set of service nodes (Routers) and receive responses asynchronously. The requests and responses are sent and received over different sockets, allowing the worker and service nodes to operate independently. This pattern allows for load balancing and fault tolerance, as the Router can distribute the requests to multiple service nodes. The Dealer-Router pattern can be used to implement a variety of distributed systems,

such as task distribution, load balancing, and parallel processing scenarios. The pattern is similar to the Request-Reply pattern, but it allows for more complex topologies and it's more suitable for scenarios where the number of workers and service nodes is dynamic.

More precisely, we have used the Router socket on every server as the “listener” socket and a list of Dealer sockets on each Client and Server, pointing to all other servers. This configuration has allowed us to “Deal” messages from any node to every other node. Chapter 4 will provide a comprehensive understanding of the system’s architecture and how the Dealers and Routers are connected.

2.3 CloudLab

CloudLab [23] is a cloud-based platform that provides researchers with access to a large-scale, distributed computing infrastructure for conducting experiments. It offers a virtual laboratory environment where researchers can design, deploy, and evaluate their experiments on a wide range of hardware and software configurations. We utilized CloudLab for our experiments to replicate a real-world scenario accurately. We were able to “rent” numerous machines inside a physical location in the US. Since machines are interconnected using a LAN, we could navigate using simple SSH.

CloudLab offers a variety of machine types to choose from in distinct geographic locations which include Utah, Wisconsin, Clemson and more. Each location is called a cluster. Clusters are made up of a varying number of different machine types. For example, the Utah cluster has 270 nodes of type m400 (64-bit ARM), 270 nodes of type m510 (Intel Xeon-D), 200 nodes of type xl170(Intel Broadwell, 10 core, 1 disk), and many more. Naturally, as this testbed has a great audience to serve, resources are limited. Large experiments with over 10 nodes often require reservations beforehand. This was the case in our experiment where we used 30 machines. Specifications on the type of machine we used and how we configured them are discussed in Chapter 5.

2.4 The Blockchain

2.4.1 Fundamentals of the Blockchain

The term Blockchain has been conceptualized and popularized by the original Bitcoin [5] by Satoshi Nakamoto. It is not clear whether Nakamoto was the original author or just a name used as an alias for the people behind Bitcoin. Regardless, we will refer to the creator/s of Bitcoin as "Nakamoto". Nakamoto not only theorized about the Blockchain but also implemented it. In simple terms, a blockchain is a decentralized, digital ledger that records transactions across a network of computers. It uses cryptography to secure and validate transactions and is organized into blocks that are linked together chronologically. Each block contains a set of transactions and a reference to the previous block, forming a chain of blocks. For a transaction to take place, one must create a public/private key pair and use the private key to send an amount to the recipient's public key. Each node in the system holds a copy of the entire ledger and waits for the new block to be advertised. A block (series of records) is only valid and can be appended to the blockchain only if it contains a "signature". That signature could be either their proof of work, such as the case with Bitcoin, their proof of stake, proof of history, or any other agreed-upon method of verifying the validity of the sender. Different blockchain networks (Bitcoin, Etherium [6], Solana [7], etc.) use different methods of validation. The most common are:

- Proof of Work (PoW): This is the original and most widely used validation method. It requires nodes (often called "miners") to perform complex mathematical calculations to validate transactions and create new blocks. The first miner to solve the calculation is rewarded with a block reward and the right to add the next block to the chain. This method is used by Bitcoin and many other cryptocurrencies.
- Proof of Stake (PoS): This method is an alternative to PoW, it doesn't require miners to perform complex calculations, instead, it relies on the nodes holding a stake in the network to validate transactions and create new blocks. The validating node is chosen at random, and the probability of being chosen is proportional to the stake that the node holds. This method is considered way more energy efficient than PoW. PoS has been recently adopted by the Ethereum blockchain [6] to establish Ethereum 2.0.
- Delegated Proof of Stake (DPoS): This is a variation of PoS, it allows token holders to vote for a set of validating nodes, and the chosen nodes will validate transactions and

create new blocks. This method is considered more efficient than PoS because the number of validating nodes is smaller than in PoS.

- Delegated Byzantine Fault Tolerance (DBFT): This method is used by the NEO blockchain, it uses a consensus mechanism based on a voting process, where nodes are elected to be consensus nodes. These nodes are responsible for validating transactions and creating new blocks.
- Proof of Authority (PoA): This is a variation of PoS, it is used in private or consortium blockchains, and it relies on a set of pre-authorized nodes, called validators to validate transactions and create new blocks. This method is considered more secure than PoS because the validators are known and trusted entities.
- Proof of Elapsed Time (PoET): This method is used in Hyperledger Sawtooth, it relies on nodes to wait for a randomly generated amount of time before they are allowed to create a new block. This approach reduces the energy consumption associated with the traditional PoW method.

In this thesis, we will use the term Blockchain to represent some of the features of a blockchain but will not explore the actual "block" aspect. We will use our validation method to append records in a non-chronologically ordered distributed ledger. Having said that, a basic understanding of the term Blockchain and what that entails is important so we can therefore understand which features and ideas of the "Blockchain" this thesis will explore.

2.4.2 Distributed Ledger Objects

Earlier we mentioned the term Ledger in a distributed system, and we will now attempt to formalize it. The work in [1] perfectly manages to formalize a variety of ledger objects. Briefly, a DLO is a data structure used to represent and manage information within a distributed ledger system. The main characteristics of a DLO is that it maintains a *sequence of records* and provides the following operations:

- *Get* – Retrieves all values of the sequence.
- *Append* – Appends a new value to the sequence.

DLOs are tamper-proof, meaning that once added to the ledger, their contents cannot be altered, and they are replicated across multiple nodes in the network. They possess the

characteristics of self-sovereignty, interoperability, and programmability which makes them a powerful tool for managing various types of assets and information such as financial transactions, digital identities, and smart contracts. DLOs are a crucial aspect of distributed ledger technology, providing a secure and transparent mechanism for the management of data and enabling the development of decentralized applications.

2.4.3 The G-Set Object

As mentioned earlier, we will be dealing with an object called a G-Set. The work in [4] provides a thorough definition of the G-Set, in an attempt to formalize a new type of data structure. The term G-Set is an abbreviation for "Grow-Only Set", meaning this data structure acts exactly like a mathematical set (contains unique values) but has the nuance of only growing. The only possible operations that can be performed on the Set are *Get* and *Add*.

- *Get* – Retrieves all values of the set.
- *Add* – Adds a new value to the set.

Conceptually the G-Set is just a data structure. However, the true challenge lies in ensuring the consistent maintenance of this data structure across multiple servers within a decentralized environment, especially in the face of malicious actors. In essence, the G-Set is a DLO that performs *Add* instead of *Append* and maintains a set rather than a sequence of records.

In our case, these values are strings, representing records for a possible application. Just like in any other blockchain, each server node has its copy of the G-Set, but unlike Bitcoin for example, the copies of the G-Set from different nodes do not have to match in sequence, just in their contents. Figure 2.1 depicts an example of a valid and non-valid final state in a network of 3 servers after all operations are marked as completed.

Valid Final G-Set State

Server 1 G-Set	Server 2 G-Set	Server 3 G-Set
Hello	!!!	World
World	World	Hello
!!!	Hello	!!!

Invalid Final G-Set State

Server 1 G-Set	Server 2 G-Set	Server 3 G-Set
Hello		World
World	World	Hello
???	Hello	!!!

Figure 2.1: Examples of Valid and Invalid G-Set State

2.5 Fault Tolerance

The general "Fault Tolerance" term refers to a system's ability to continue operating without interruption or with minimal degradation when one or more of its components fail. Let us now discuss what this means in distributed systems.

2.5.1 Fault Tolerance in Distributed Systems

Fault tolerance in Distributed Systems is the ability of a distributed system to continue functioning correctly in the presence of failures [29]. It is a crucial aspect of distributed systems, as it ensures that the system can continue operating despite various failures such as hardware failures, software bugs, and malicious attacks.

This can be achieved through techniques such as *replication*, *redundancy*, and *consistency* protocols. Replication involves creating multiple copies of data and distributing them across different nodes in the system to ensure that the data is still available in the event of a failure. Redundancy refers to using multiple components in the system so that if one fails, the others can take over. Consistency ensures that the nodes in the system maintain the same state, despite the presence of failures.

2.5.2 Byzantine Generals Problem

The *Byzantine Generals Problem* [8] is a concept in distributed computing that describes the challenge of achieving agreement among a group of unreliable nodes. It is named after a hypothetical situation in which a group of generals must decide whether to attack or retreat, but some of the generals may be traitors who are trying to sabotage the decision-making process. The problem illustrates the difficulties of achieving agreement in a distributed system when nodes can fail in arbitrary ways. The problem was first introduced in a paper titled "The Byzantine Generals Problem" by Leslie Lamport, Robert Shostak, and Marshall Pease in 1982 [8].

Suppose we have several generals each with their army, surrounding a city they want to capture. The armies have great physical distance between them therefore direct communication is not possible. They all know that a coordinated attack will be successful but any attempt at an uncoordinated attack will lead to defeat.

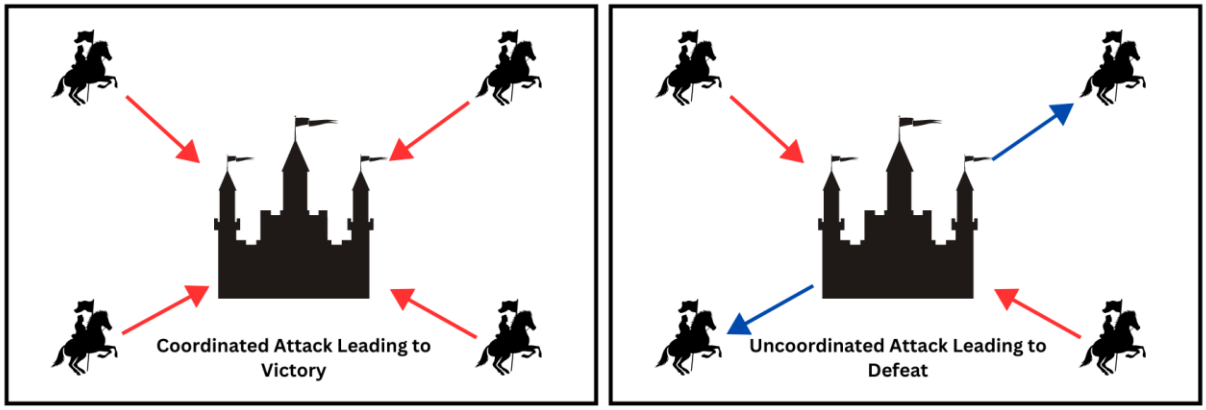

Figure 2.2: Byzantine Generals Problem Inspired by [12]

One might suggest that a messenger on horseback could inform all the generals to "Attack at dawn". In that case, the general who sent the messenger would have to wait for a reply from every other general. This scenario introduces 2 problems:

- The messenger gets captured either on the way to deliver the initial message or the reply.
- The messenger gets captured and replaced by an impostor who delivers fake messages to sabotage the attack.

In either case, the attack will fail. This scenario begs the question, how can the generals communicate and reach a consensus in this scenario?

Over the years, many solutions have been proposed to address the Byzantine Generals' problem. One of the most well-known solutions is the Practical Byzantine Fault Tolerance (PBFT) algorithm, which was introduced by Miguel Castro and Barbara Liskov in 1999. [9]. While PBFT and other solutions are effective in addressing the Byzantine Generals problem in practical scenarios, the problem is still considered an open research problem in distributed computing, and researchers continue to work on developing more efficient and scalable solutions. Therefore, it can be said that the Byzantine Generals' problem is not completely solved, but there are practical solutions that can be used in real-world scenarios.

2.5.3 Byzantine Fault Tolerance

We have now seen the Byzantine Generals Problem and have a basic idea of what Byzantine Faults are in distributed systems. Let us now define the term Byzantine Fault.

The Byzantine fault is a type of fault that can occur in a distributed system, in which a node behaves arbitrarily and unpredictably. This can include providing incorrect or inconsistent information to other nodes, failing to respond to requests, or otherwise deviating from the system's expected behavior. Byzantine faults are particularly challenging to handle in distributed systems, as they can lead to incorrect or inconsistent system behavior and can be difficult to detect and diagnose.

Some of the solutions proposed rely on probability (Bitcoin), and the unlikeliness of some events. Other solutions, however, work perfectly under some predefined conditions. Such is the case with our considered algorithms, which guarantee a solution if the number of faulty nodes in the network is at most equal to $(n\text{-}1)/3$, with n representing the number of nodes in the system.

2.5.4 Bracha's Reliable Broadcast Algorithm

As mentioned earlier, our considered algorithms are Byzantine-Fault Tolerant only if the number of faulty nodes *f* in the network is at most equal to (*n*-1)/3. This threshold is directly derived from Bracha's Reliable Broadcast Algorithm [10] which is an essential part of our algorithm. Bracha's Reliable Broadcast Algorithm is a Byzantine Fault Tolerance (BFT) algorithm that ensures reliable messages broadcast in a distributed system. It was first proposed by Bracha in 1987 [10] and allows multiple messages to be broadcast in a fault-tolerant way. The algorithm ensures that all correct processes receive the same set of messages and that no correct process receives any message that was not broadcast. Bracha's algorithm is a basic building block for many BFT protocols, and it is a pioneer work in the field of BFT. Bracha's algorithm is especially useful in systems where multiple messages must be broadcast in a fault-tolerant way, and where ensuring the reliability of the broadcast is critical for the correct operation of the system. The work in [11] provides an excellent and basic implementation along with an explanation. Let us present the algorithm and a brief breakdown using Figure 2.3.

```
// leader with input v
send <v> to all parties
// Party j (including the leader)
echo = true
vote = true
on receiving <v> from leader:
   if echo == true:
      send <echo, v> to all parties
      echo = false
on receiving <echo, v> from n-f distinct parties:
   if vote == true:
      send <vote, v> to all parties
      vote = false
on receiving <vote, v> from f+1 distinct parties:
    if vote == true:
      send <vote, v> to all parties
      vote = false
on receiving <vote, v> from n-f distinct parties:
    deliver v
```

Figure 2.3: Pseudocode for Bracha's Reliable Broadcast Algorithm by [11]

1. To begin with, we require each party to echo only one message and wait for $n-f$ echo messages before voting for the leader. This forces the leader to transmit just one value. Since any two sets of $n-f$ must overlap by at least $f+1$ parties, two distinct non-faulty parties can't support negative values.
2. Second, we make sure that all non-faulty supply the same value if one non-faulty does. We do this by mandating that a party transmit just one vote after viewing either $n-f$ echo messages or $f+1$ votes. As a result, if any party receives $n-f$ votes, all non-faulty parties will receive $n-2f \geq f+1$ votes.

The algorithm guarantees that all correct processes will receive the same set of messages and that no correct process will receive any message that was not broadcast. The algorithm's proven correctness and asynchronous nature make it perfect for the system we aimed at creating and evaluating, thus is an essential part.

Chapter 3

The Algorithms

3.1 Consensus-free Eventual Consistency

To understand the algorithms we implemented, we first must gain a grasp on the way consistency is achieved. As mentioned earlier, some popular distributed systems use consensus protocols to maintain their correctness. However, the algorithms in [4], achieve what is called “Eventual consistency” without relying on any consensus protocol.

In a storage system, Eventual Consistency [30] is a type of weak consistency where, if no further modifications are made to an object, all upcoming accesses to that object will ultimately return the value that was most recently modified. Communication lags, system stress, and the quantity of clones all affect how long inconsistency persists. A well-known system that uses eventual consistency is the domain name system (DNS). DNS uses time-controlled caches and a defined pattern to disseminate name changes, making sure that ultimately all clients will see the change.

In this thesis we used Bracha's Reliable Broadcast (BRB) Algorithm for the communication between servers. As a direct result of the use of BRB, a hard boundary on the number of Byzantine servers in the system is set. That being $f=(n-1)/3$, where n is the number of machines running the algorithm

3.2 Eventually Consistent Byzantine-tolerant Distributed G-Set Object

In this section, the pseudocode for the Eventually consistent Byzantine-tolerant Distributed G-Set Object (BDSO) is provided (as seen in [4]) and explained.

3.2.1 Client API

```
1: Init: c ← 0
2: function GS.get( ) . Invocation event
3: c ← c + 1
4: send request get(c, p) to 3f + 1 different servers
5: wait responses getResp(c, i, Si) from 2f + 1 different
servers
6: S ← {r : record r is in at least f + 1 sets Si}
7: return S . Response event
8: function GS.add(r) . Invocation event
9: c ← c + 1
10: send request add(c, p, r) to 2f + 1 different servers
11: wait responses addResp(c, i, ack) from f + 1 different
servers
12: return ack
```

Figure 3.1: Client API and algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object GS [4]

The algorithm in Figure 3.1 implements the client API and algorithm for an eventually consistent, Byzantine-tolerant distributed G-Set object GS. The algorithm consists of two functions: *GS.get()* and *GS.add()*, which can be invoked by the client.

The algorithm begins by initializing a counter variable c to 0 (line 1). This counter is used to distinguish each client request. A client's identity in combination with its message counter produce a unique combination. When the *GS.get()* function is invoked (line 2), the counter c is incremented (line 3) and a request is sent to $3f+1$ different servers to retrieve the current

state of the G-Set object (line 4). The client then waits for responses from 2*f*+1 different servers (line 5). Once enough responses are received, the responses are merged to form a set S of all records that are in at least *f*+1 of the sets S (line 6). The set S is then returned to the client (line 7). Similarly, when the *GS.add()* function is invoked (line 8), the counter c is incremented (line 9) and a request is sent to 2*f*+1 different servers to add a new record r to the G-Set object (line 10). The client then waits for responses from *f*+1 different servers (line 11). Once enough responses are received, the client returns an acknowledgment to indicate that the record has been added (line 12).

This algorithm is designed to be Byzantine-tolerant, meaning that it can tolerate arbitrary failures or malicious behavior from up to $f<n/3$ of the total number of servers in the system. The algorithm achieves eventual consistency by ensuring that updates are propagated to at least *f*+1 servers, and by merging responses from at least *f*+1 servers to form a consistent view of the G-Set object, plus the use of BRB on the server side, as explained next.

3.2.2 Server API

```
1: Init: Si ← ∅
2: receive (get(c, p)) from process p . Signature of p is
validated
3: send response getResp(c, i, Si) to p
4: receive (add(c, p, r)) from process p . Signature of p
is validated
5: if (r /∈ Si) then
6: BRB-broadcast(propagate(i, add(c, p, r)))
7: wait until r ∈ Si
8: send response addResp(c, i, ack) to p
9: upon (BRB-deliver(propagate(j, add(c, p, r)))) do .
Signatures of j and p are validated
10: if (r /∈ Si) and (add(c, p, r) was received from f + 1
different servers j) then
11: Si ← Si ∪ {r}
```

Figure 3.2: Server algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object

The algorithm in Figure 3.2 implements the server-side algorithm for an eventually consistent, Byzantine-tolerant distributed G-Set object (BDSO). The algorithm consists of two operations: handling a get request (lines 2-3) and handling an add request (lines 4-11).

The algorithm begins by initializing the server's set Si to the empty set (line 1). When a get request is received from process p (line 2), the server validates the signature of p and responds to the request with a *getResp* message containing the current state of the set Si and the request's counter value c (line 3). When an add request is received from process p (line 4), the server first validates the signature of p. If the record r being added is not already in the server's set Si (line 5), the server initiates a Byzantine Reliable Broadcast (BRB) to propagate the add request to all other servers in the system (line 6). The server then waits until the record r is present in its own set Si (line 7). Once the record r is present, the server sends a response addResp message containing the request's counter value c and an acknowledgment to process p (line 8). When the server receives a BRB-deliver message propagating the add request from another server j (line 9), it validates the signatures of both j and p. If the record r is not yet present in the server's set Si, and the add request was received from at least f+1 different servers, including server j (line 10), the server adds the record r to its set Si (line 11).

3.3 Two-Atomic Adds

We now provide a use case of the BDSO. A system that uses the principles of the BDSO, and with slight modification solves a widespread problem in distributed systems.

3.3.1 The Atomic Adds Problem

The *Atomic Appends* problem is a distributed ledger interconnection problem that requires appending several records in their corresponding distributed ledgers, such that either all records are appended, or none is appended to any distributed ledger. The problem was introduced in [2] and was formulated and solved in the presence of Byzantine servers and clients in [3]. Two records are *mutually dependent* if one record can only be appended to its intended distributed ledger if the other record is appended to its intended distributed ledger.
The 2-Atomic Appends problem refers to the case where two clients have mutually dependent records. The problem requires that records be appended atomically in their corresponding distributed ledgers to guarantee both safety and liveness properties.

Naturally, as we have provided the algorithm for a BDSO, we will be solving a variation of the Atomic Appends problem, which is the *2-Atomic-Adds* problem [4]. The Atomic Appends problem and the Atomic Adds problem are similar in that they both involve adding records to a data structure in a distributed system. However, there are some key differences between the two problems. In the Atomic Appends problem, the records are added to a BDLO (Byzantine fault-tolerant Distributed Ledger Object), which is a replicated log with strong consistency guarantees. The goal is to ensure that all correct clients agree on the order in which the records are appended to the log. The safety requirement is that no two correct clients can append conflicting records, and the liveness requirement is that every correct client eventually appends their record.

In contrast, the Atomic Adds problem involves adding records to multiple BDSOs (Byzantine fault-tolerant Distributed Set Objects). The goal is to ensure that all correct clients add their records atomically to their corresponding BDSO, meaning that the records are either all added, or none are added. The safety requirement is that a correct client can only add their record if the record of the other client is also added, and the liveness requirement is that if both clients are correct, then both records are added eventually. In terms of implementation, the Atomic Appends algorithm uses a BDLO to ensure strong consistency, while the Atomic Adds algorithm uses the BDSO implementation from Section 3 of the paper. The solution in [4] uses a special-purpose distributed ledger called Smart BDLO to aggregate and coordinate the append of multiple records.

For a better understanding of the problem that the Smart BDSO is solving, suppose the following scenario which is depicted in Figure 3.3:
Person A: Wants to buy a car.
Person B: Wants to sell the same car.
Car ownership documents are stored in **BDSO 1**.
Monetary transactions can be done using a cryptocurrency hosted in **BDSO 2**.
Persons A and B **do not trust each other**.
How can the exchange of ownership and money take place with both parties satisfied?

This problem can only be solved by using an intermediary Smart BDSO, which we will just call SBDSO. Both parties agree to send their transaction request to the SBDSO and if the SBDSO notices 2 matching records, it is responsible for adding the transaction from Person

A to BDSO 2 (payment) and the transaction from Person 2 to BDSO 1 (transfer of ownership). The work in [2] has proved that the use of an intermediary network such as the SBDSO is necessary to solve the *Atomic Appends/Adds problem*.

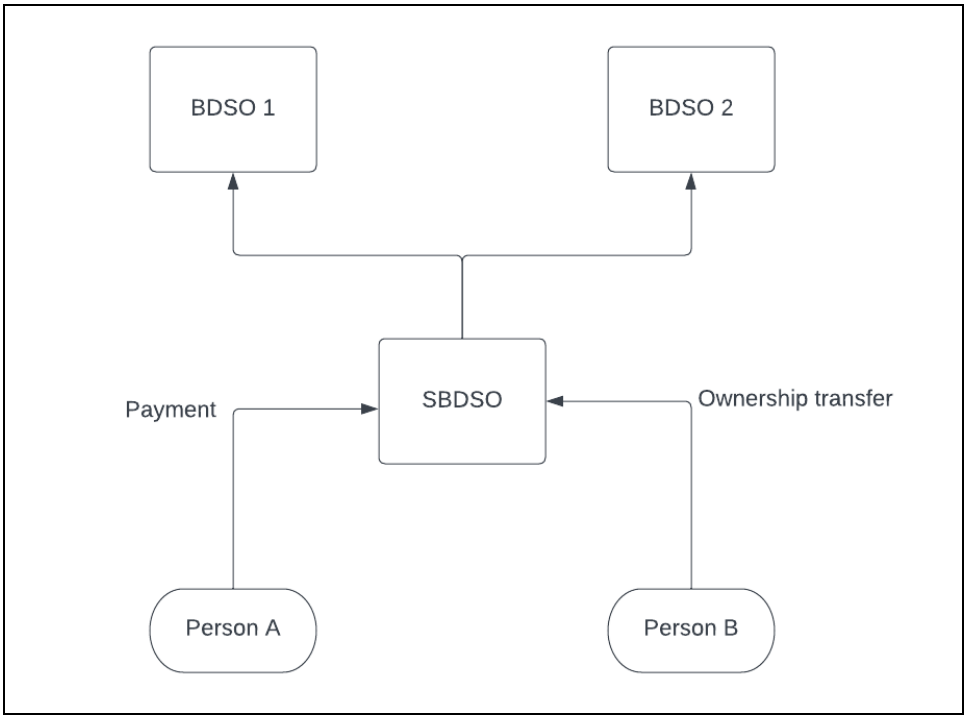

Figure 3.3 Purchase of a Car using an SBDSO

3.3.2 Client API

```
1: function AtomicAdds(p, {p, q}, rp, GSp, rq)
2: GS.add(hp, {p, q}, rp, GSp, rqi)
3: return ack
4: // Client p will know the Atomic Adds operation was
completed successfully when it receives notifications
from f + 1 different SBDSO servers.
```

Figure 3.4: API for the 2-AtomicAdds of records rp and rq in BDSOs GSp and GSq by clients p and q, respectively, using SBDSO GS. Algorithm for Client p

The algorithm in Figure 3.4 presents an API for implementing the 2-AtomicAdds problem, where two clients p and q, with mutually dependent records rp and rq, respectively, add records to their corresponding BDSOs GSp and GSq, such that either all records are added, or

none is added. The algorithm uses the SBDSO GS to ensure consistency between the clients and includes a function called AtomicAdds that takes as input the client identifier p, the set of correct clients {p, q}, the record rp to be added to GSp, and the record rq to be added to GSq. The function adds the records to their respective BDSOs and returns an acknowledgment to the client. The successful completion of the operation is determined by receiving notifications from *f+1* different SBDSO servers.

3.3.3 Server API

```
1: Init: Si ← ∅
2: receive (add(c, p, r)) from process p . Signature of p
is validated
3: if (r /∈ Si) then
4: BRB-broadcast(propagate(i, add(c, p, r)))
5: wait until r ∈ Si
6: send response addResp(c, i, ack) to p
7: upon (BRB-deliver(propagate(j, add(c, p, r)))) do .
Signatures of j and p are validated
8: if (r /∈ Si) and (add(c, p, r) was received from f + 1
different servers j) then
9: Si ← Si ∪ {r}
10: if (r.v = hp, {p, q}, rp, Lp, rqi) and
11: (∃r 0 ∈ Si : r 0 .v = hq, {p, q}, rq, Lq, rpi) then
12: Lp.append(rp); Lq.append(rq)
13: Notify clients p and q that records rp and rq have
been appended to Lp and L
```

Figure 3.5: Smart Byzantine-tolerant DSO; Only the code for the add operation is shown; Code for Server i

The algorithm in Figure 3.5 presents the add operation in a Smart Byzantine-tolerant DSO system. The code for the get operation is omitted in this section since it is identical to the BDSO code provided in Section 3.2.2. It receives a request to add a record (c, p, r) from a process p. The record is broadcasted to other servers in the network using a Byzantine Reliable Broadcast (BRB) protocol. Once the record is added to the local state of the server, it sends an acknowledgment to the requesting process. When a server receives a BRB-deliver message containing the same record from another server, it validates the signatures and adds the record to its local state if it hasn't already been added and has been received from *f*+1 different servers.

Chapter 4

Implementation

4.1 Implementation Decisions

The entire codebase of this thesis is stored in a GitHub repository [14]. We tried to separate each version of the project into its project folder. This becomes clear in the following section regarding the project structure. The local and remote implementations of each version were also separated.

The experimental evaluation was mainly focused on the BDSO, denoted by the name folder name “BFT-Distributed-G-Set”, as this is the foundation and focus of this thesis. This will become even more apparent from the scripts in the “BFT-Distributed-G-Set” folder, aimed at extensively evaluating this part of the thesis.

Another important decision we made when developing the codebase for this thesis is omitting the signature validation mechanism. The APIs presented earlier denote the use of a signature validation mechanism aimed at ensuring the validity of the servers. This thesis' aim, however, is to evaluate the performance of the algorithm and therefore concluded that the signature validation mechanism is a trivial part that can be left out without hindering the code's correctness. It should be noted that if the algorithm is to ever be commercially used, the signature verification mechanism should be included.

The main idea of the way we handle clients and servers is by creating an object for each client or server that holds all its information including but not limited to its: hostname, receive port, amounts of messages received, socket references, and many more. More details are provided in later sections. Then, the object reference is passed around to different functions that send, receive, and handle requests, updating the object's state. Knowing the basic idea behind the codebase will allow us to easily understand some code snippets that follow. For this section, to explain the code, we assume that the code is running on a Local Area Network of machines with hostnames node0 – nodeX, where X+1 is the number of available machines. More details on the evaluation environment are provided in the next chapter.

4.2 Project Structure

We divided the project into smaller projects, each being the building block of the next. A natural evolution in the codebase can be observed, as every subsequent section has some additions, changes, and fixes. The entire project structure is as follows (excluding some automation scripts that will be seen in the GitHub repository):

Figure 4.1 showcases the directory structure of the thesis project. The Bracha-Reliable-Broadcast-Local folder contains a server folder and was created as an experimental scenario on localhost for us to evaluate the correctness of our implementation of Bracha's Reliable Broadcast algorithm.

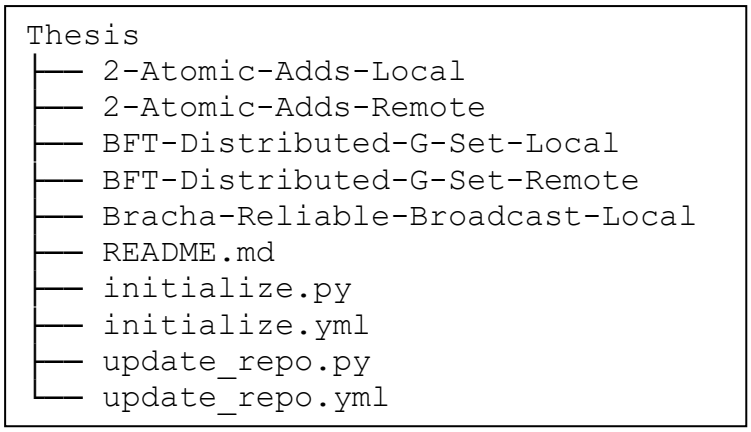

Figure 4.1: Codebase Structure

The BFT-Distributed-G-Set-Local folder contains the local implementation of our main algorithm, once again aiming to evaluate correctness, fix bugs, etc. The BFT-Distributed-G-Set-Remote folder contains the focus of this thesis, including a client and server folder containing the code for the clients and servers respectively. A lot more files will be found in this folder in comparison to the rest, relating to the experimental evaluation. More details will be provided in the next chapter.

Finally, the 2-Atomic-Adds-Local and 2-Atomic-Adds-Remote contain an implementation of a system solving the 2-Atomic-Adds problem in a local and remote setting. These folders contain the code for the clients, the code for 2 networks of BDSO servers, and the code for a network of Smart-BDSO servers, commonly referred to in this thesis as SBDSO.

4.3 BFT-Distributed-G-Set System

We will now provide the most important parts of the implementation of the BFT-Distributed-G-Set System as shown in Figure 4.2. We only present the remote implementation as the local implementation was only used for debugging purposes.

Some utility scripts are also included in the project folder, just used to streamline the evaluation process. These scripts perform a variety of tasks, such as running automated tests, generating the results, and checking the state of the experiment.

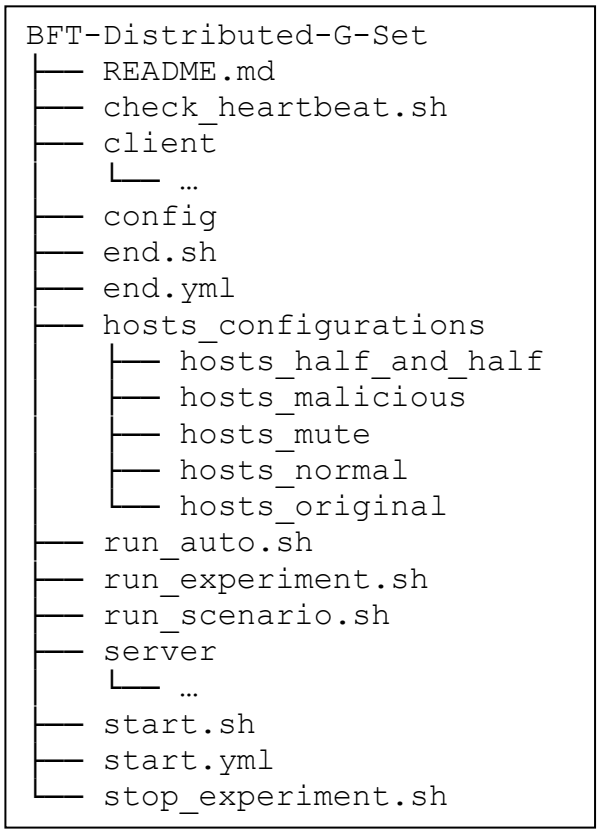

Figure 4.2: BDSO Code Structure

4.3.1 Topology

Before we look at any code it is important to examine the topology of the system, thus making it easier to understand the code. As mentioned earlier, we used entirely ZMQ for the communication between nodes, particularly the DEALER-ROUTER paradigm.

The servers used a combination of DEALER sockets and ROUTER sockets. Each server has one DEALER socket for sending messages to all the other servers, necessary for Bracha's Reliable Broadcast Algorithm, and another ROUTER socket with the responsibility of listening to messages from all its peers and clients. The clients solely used DEALER sockets to connect to the servers.

Figure 4.3 depicts the topology of the system, socket types, and how they are connected among the servers and clients.

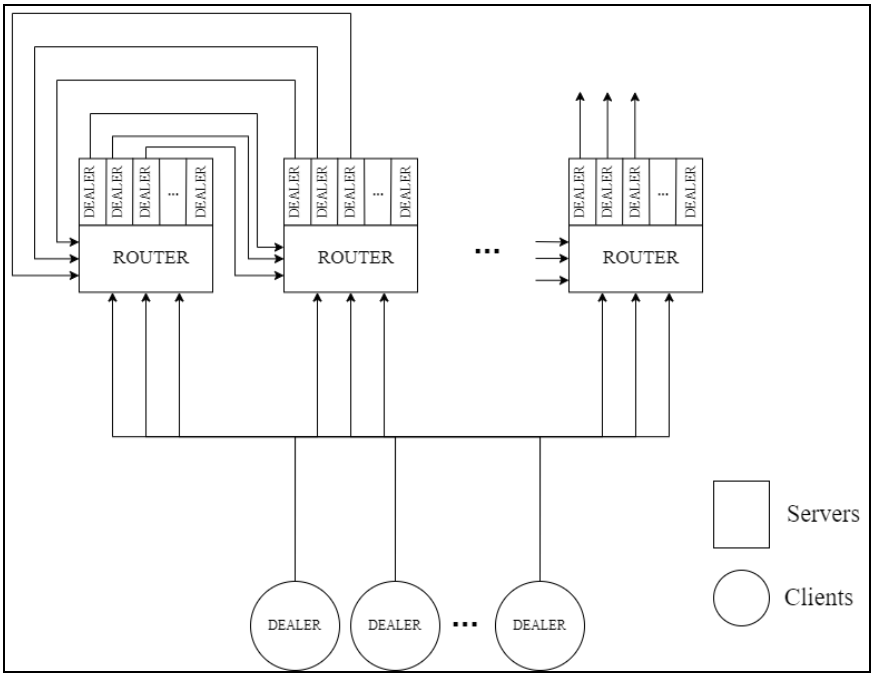

Figure 4.3: BDSO Topology

4.3.2 Hosts and Configuration

Inside the project folder, we have two important files that determine the behavior of the system, the “hosts” file and “config” file.

The hosts file provides the blueprint for the experiment. Essentially, determines the behavior of each node in the system. When performing the experiments, we prepared a variety of hosts configurations and placed them inside the `hosts_configurations` folder. The script running the experiments would copy a hosts file from that directory and place it in the program’s working directory, perform the experiment, and then delete it to make way for the next. This is an example of a hosts file.

```
[master]
node0

[clients-automated]
node1
node2

[servers-normal]
node3
node4
node5

[servers-mute]
node6

[servers-malicious]
node7
node8
node9

[servers-half_and_half]
```

Figure 4.4: Hosts File

The "config" file, on the other hand, states the number of threads that each node will start and the first port that the first thread will listen to (if the node is representing a server).

```
DEFAULT_PORT=5555
NUM_THREADS=2
```

Figure 4.5: Config File

Figure 4.5 showcases an example of the config file. If node1 is under the `[servers-normal]` tag, and `NUM_THREADS=2` and `DEFAULT_PORT=5555`, node1 will start 2 threads with normal server behavior, one listening on port 5555 and the other on 5556.

4.3.3 Client Implementation

The client's main task is sending the various GET and ADD requests while also verifying and counting the matching replies. This can be done in either an interactive or automated way as will become apparent in the code. Let us now present the structure and some key code snippets from the client code, of which the directory structure is presented in Figure 4.6.

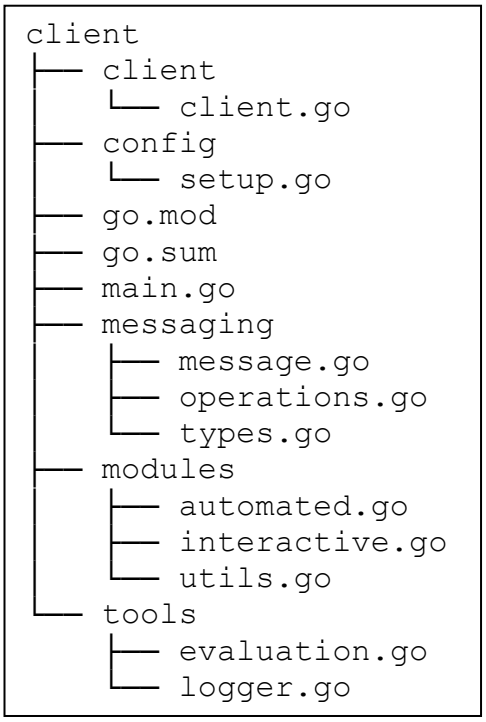

Figure 4.6: BDSO Client Code Structure

Inside the main directory, we have a folder for each package (standard Golang practice [16]) and also the main.go file along with Golang's project organization files: go.mod and go.sum. We now provide a brief description of each package and expand with code snippets where we deem necessary.

Client Package

The client package is used to declare the client object and all its attributes and data, along with a creation function. Figure 4.7 provides the function declaration for the creation of the client object.

```
func CreateClient(id string, servers []config.Node, zctx
*zmq.Context) *Client {…}
```

Figure 4.7: BDSO CreateClient Function Declaration

```
type Client struct {
     Id              string
     Zctx            *zmq.Context
     Poller          *zmq.Poller
     Message_counter int
     Servers         map[string]*zmq.Socket

     // Experimental evaluation statistics
     TOTAL_GET_TIME int
     TOTAL_ADD_TIME int
     REQUESTS       int
}
```

Figure 4.8: BDSO Client Struct

Figure 4.8 shows the Client struct which contains data such as

- The client's identity, which can be a plain string, such as a name.
- The ZeroMQ context.
- ZeroMQ poller, which is used to retrieve incoming messages.
- A map of strings to ZeroMQ sockets representing the servers' hostnames and sockets to send messages.
- Variables for keeping track of the average add and get time.

Config Package

The Config package comprises functions for initialization and file parsing, and it declares the Node struct, along with the global variables *n* and f. These variables represent the total number of servers and the number of Byzantine servers, respectively.

```
type Node struct {
     Host string
     Port string
}

var (
     N = 0
     F = 0
)
```

Figure 4.9: BDSO config.go

Messaging Package

The messaging package contains a declaration of a Message object, the message tags, but mainly the operations.go file. Inside this file, we provide the implementation of the *GET* and *ADD* operations. Figure 4.10 shows the code for the Get function.

```
func Get(c *client.Client) (string, time.Duration) {
     tools.Log(c.Id, "Called GET")
     c.Message_counter++
     start := time.Now()
     sendToServers(c.Servers, []string{GET,
strconv.Itoa(c.Message_counter)}, 3*config.F+1)

     replies := make(map[string]string)
     tools.Log(c.Id, "Waiting for valid GET_REPLY")
     for {
          sockets, _ := c.Poller.Poll(-1)
          for _, socket := range sockets {
               s := socket.Socket
               msg, _ := s.RecvMessage(0)
               if msg[1] == GET_RESPONSE {
                    replies[msg[0]] = msg[2]
               }
          }
          r := countMatchingReplies(replies)
          if len(r) > 0 {
               elapsed := time.Since(start)
               tools.Log(c.Id, "GET completed in:
"+elapsed.String())
               return r, elapsed
          }
     }
}
```

Figure 4.10: BDSO Client GET Function

The *Get* function takes in the reference of the Client as a parameter and returns a string representing the entire G-Set, as well as the time it takes for the client to receive the required number of matching replies. The client first sends the GET request to 3*f*+1 servers and then waits until the poller has gathered any replies. If it has and the message tag is GET_RESPONSE, the message is added to a map with the corresponding sender id. Finally, the function countMatchingReplies makes sure we have over *2f*+1 replies and at least *f*+1 of those, are the same.

Modules Package

The modules package essentially contains the “frontend” code that calls the backend API functions declared in operations.go. It provides the implementation of functions such as StartAutomatic and StartInteractive, indicating the type of session that the client can start. The automatic module is used for testing the system, whereas an interactive session introduces a shell-like environment that allows the user to interact directly with the system. Figure 4.11 displays an example of how the shell-like environment will look after providing a user id and calling an ADD and GET operation. User inputs are highlighted.

```
Your ID
> loukas
ID set to 'loukas'

| loukas | Established connection with tcp://localhost:5500
| loukas | Established connection with tcp://localhost:5501
| loukas | Established connection with tcp://localhost:5502
| loukas | Established connection with tcp://localhost:5503
| loukas | Established connection with tcp://localhost:5504
Type 'g' for GET, 'a' for ADD or 'e' for EXIT
> a
Record to append > hello_world
| loukas | Called ADD(hello_world)
| loukas | Waiting for f+1 ADD replies
| loukas | ADD completed in: 3.290809ms. Record
{hello_world} appended
Type 'g' for GET, 'a' for ADD or 'e' for EXIT
> g
| loukas | Called GET
| loukas | Waiting for valid GET_REPLY
| loukas | GET completed in: 798.102µs
| loukas | {hello_world}
Type 'g' for GET, 'a' for ADD or 'e' for EXIT
>
```

Figure 4.11: BDSO User Interface

Tools Package

The tools package is perhaps the simpler and smallest of all the other packages. It provides some logging functionality as well as utility functions that keep track of the statistics during our experimental evaluation.

Main

Finally, the main function is provided outside of any package denoted at the top of the main.go file with the statement “package main”. This is the entry point of the client code. Its responsibilities include parsing the hosts file, creating the ZeroMQ context, initializing the *n* and *f* variables, and initializing the log file. The main function also provides a flag system for passing in client arguments regarding automation. The “auto” boolean flag states whether the session will be automated and if so, the number of clients and requests each one of them will make, can be passed using the “clients” and “reqs” flags respectively. A possible execution command is shown in Figure 4.12.

```
…/client$ go run main.go -auto clients 3 -reqs 5
```

Figure 4.12: BDSO Client Run Command

And the main function can be seen in Figure 4.13.

```
func main() {
        tools.LOGGING = true
        tools.ResetLogFile()

        wd := "/users/loukis/Thesis/BFT-Distributed-G-Set-
Remote"
        _, client_threads := config.GetPortAndThreads(wd +
"/config")
        zctx, _ := zmq.NewContext()

        var auto bool
        var reqs int
        var clients int

        flag.BoolVar(&auto, "auto", false, "Automated")
        flag.IntVar(&reqs, "reqs", 5, "Amount of requests")
        flag.IntVar(&clients, "clients", client_threads,
"Clients”)

        flag.Parse()

        if auto {
                modules.StartAutomated(zctx, clients, reqs)
                return
        }
        modules.StartInteractive(zctx)
}
```

Figure 4.13: BDSO Client Main Function

4.3.3 Server Implementation

The server's main task is listening to clients' *GET* and *ADD* requests and serving them. The server code can emulate various behaviors, acting either normally or ill-intended. Each server is also responsible for running its code of Bracha's Reliable Broadcast code. When the BDSO algorithm was presented in the previous chapter, Bracha's Reliable Broadcast algorithm was referred to as a service, outside of the server code. For our implementation, we chose to include this code inside each server for performance and simplicity.

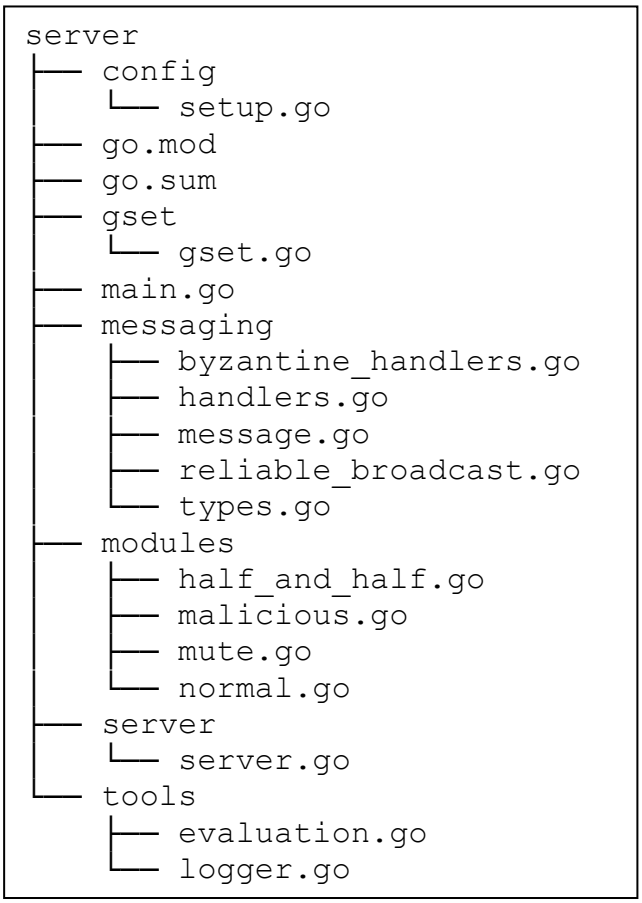

Figure 4.14: BDSO Server Code Structure

Similarly, to the client code, the server code has a package-oriented structure. Let us now briefly describe the packages, with some code snippets wherever deemed necessary. The directory structure for the server code is shown in Figure 4.14.

Config Package

The server config package is very similar to the client's config package as it declares the *n* and *f* variables and some file parsing functions. Mainly the hosts file parsing function which lets the current machine know who its peers are and what is their behavior.

GSet Package

The gset package contains the implementation of the G-Set object, the data structure that holds all our records. We chose to leave the G-Set as a data structure in the machine's memory, and not keep a persistent state because of the experimental nature of this thesis. The G-Set is simply represented by a native Golang map [17]. It is a map of strings mapping other strings. Figure 4.15 depicts the G-Set creation function.

```
// Create gset
func Create() map[string]string {
     return make(map[string]string)
}
```

Figure 4.15: G-Set Creation Function

As our system has high volumes of not only reading but writing to the data structure, we chose to use maps because of their constant lookup and add times. The values of the map are the actual records and the keys are their corresponding SHA512 representation [19]. The way we encode the records is shown in Figure 4.16.

```
// Utility function to convert string record to
// a sha516 string value to be used as key
func string_to_sha512(s string) string {
     h := sha512.New()
     h.Write([]byte(s))
     sha512_hash := hex.EncodeToString(h.Sum(nil))
     return sha512_hash
}
```

Figure 4.16: String to SHA512 Conversion Function

Along with the declaration and creation of the G-Set, this package provides the methods for Adding to the G-Set, checking for the existence of elements, and conversion of the G-Set to a human-readable string.

Server Package

The server package is used to declare the server object and all its attributes and data, along with a creation function. The function declaration for creating a server is shown in Figure 4.17.

```
func CreateServer(me config.Node, peers
[]config.Node) *Server {…}
```

Figure 4.17: BDSO CreateServer Function Declaration

Figure 4.18 depicts the server struct which contains many fields. We only explain the most important.

```
type Server struct {
     Zctx           *zmq.Context
     Id             string
     Peers          map[string]*zmq.Socket
     Receive_socket *zmq.Socket
     Host           string
     Port           string
     Gset           map[string]string

     // Used for reliable broadcast
     My_init    map[string]bool
     My_echo    map[string]bool
     My_vote    map[string]bool
     Peers_echo map[string]bool
     Peers_vote map[string]bool

     …
}
```

Figure 4.18: BDSO Server Struct

- Zctx: The ZeroMQ context
- Host: The hostname of the machine
- Port: The port that the server is listening to for messages
- Receive_socket: The ZeroMQ socket variable used to retrieve the messages
- Id: Combination of host and port for a unified identifier
- Gset: Each server's copy of the G-Set object
- My_init, My_echo, My_vote, Peers_echo, Peers_vote: Pools of messages used by the reliable broadcast algorithm.

Messaging Package

The messaging package contains anything related to sending and receiving messages. The types.go and message.go files have the same utility as in the client code. Inside the handlers.go file we have functions that handle incoming messages.

```
func HandleMessage(s *server.Server, msg []string) {
     message, err := ParseMessageString(msg)
     if err != nil {
          tools.Log(s.Id, "Error msg: "+strings.Join(msg, "
"))
          return
     }
     if message.Tag == GET {
          tools.Log(s.Id, "Received "+message.Tag+" from
"+message.Sender)
     } else {
          tools.Log(s.Id, "Received "+message.Tag+"
{"+strings.Join(message.Content, " ")+"} from
"+message.Sender)
     }

     if message.Tag == GET {
          handleGet(s, message)
     } else if message.Tag == ADD {
          message.Content[0] = message.Sender + "." +
message.Content[0]
          handleAdd(s, message)
     } else if strings.Contains(message.Tag,
BRACHA_BROADCAST) {
          handleRB(s, message)
     }

}
```

Figure 4.19: BDSO Server HandleMessage Function

The `HandleMessage` function which is shown in Figure 4.19, is responsible for parsing the incoming messages and calling the right handler functions such as `handleGet`, `handleAdd`, and `handleRB` (Reliable Broadcast).

```
func handleGet(receiver *server.Server, message Message) {
	response := []string{message.Sender, receiver.Id, GET_RESPONSE,
gset.GsetToString(receiver.Gset, false)}
	receiver.Receive_socket.SendMessage(response)
	tools.Log(receiver.Id, GET_RESPONSE+" to "+message.Sender)
}

func handleAdd(receiver *server.Server, message Message) {
	if !gset.Exists(receiver.Gset, message.Content[0]) {
		ReliableBroadcast(receiver, message)
	} else {
		response := []string{message.Sender, receiver.Id,
ADD_RESPONSE, message.Content[0]}
		receiver.Receive_socket.SendMessage(response)
	}
}
```

Figure 4.20: BDSO Server handleGet Function

The handleGet function, shown in Figure 4.20, simply returns the current G-Set to the sender. The handleAdd function checks whether the record to Add is already inside the G-Set. If this is the case, the server just responds with a "success" message to the sender. If not, the ReliableBroadcast function is called to inform the servers' peers about the message.

```
// Function to interact with reliable_broadcast
func handleRB(receiver *server.Server, message Message) {
	// prepare the response
	response := []string{message.Content[0], receiver.Id,
ADD_RESPONSE, message.Content[1]}

	// check if the record already exists
	record := message.Content[1]
	if gset.Exists(receiver.Gset, record) {
		receiver.Receive_socket.SendMessage(response)
		return
	}

	// pass message through the true handler
	// if rb is complete (aka delivered), add message to gset
	delivered := HandleReliableBroadcast(receiver, message)
	if delivered {
		gset.Add(receiver.Gset, message.Content[1])
		receiver.Receive_socket.SendMessage(response)
		tools.Log(receiver.Id, "Appended record
{"+message.Content[1]+"}")
		return
	}
}
```

Figure 4.21: BDSO Server handleRB Function

Figure 4.21 depicts the `handleRB` function. The responsibility of this function is interacting with the functions of the reliable_broadcast.go file, where Bracha's Reliable Broadcast algorithm is implemented. When any message is received, and its tag contains the keyword "BRACHA" it is passed through `handleRB`. Similarly, to `handleAdd`, if the record exists, the function returns a "success" message to the sender, tagged with "ADD_RESPONSE". If not, the message is passed through the `HandleReliableBroadcast` function, which checks whether the reliable broadcast for that message is complete, and if it is, the record is added to the G-Set.

Let us now explain the reliable_broadcast.go file, the implementation of Bracha's Reliable broadcast algorithm, one of the most integral parts of this thesis. Figure 4.22 shows the `ReliableBroadcast` function as well as the header for the `HandleReliableBroadcast` function.

```
// Leader, the one who initializes the module
func ReliableBroadcast(leader *server.Server, message
Message) {
     content := append([]string{message.Sender},
message.Content...)
     tag := BRACHA_BROADCAST_INIT
     v := CreateMessageString(tag, content)
     // leader with input v
     sendToAll(leader, v)
}

// Called from every server receiving RB messages
func HandleReliableBroadcast(receiver *server.Server, v
Message) bool {…}
```

Figure 4.22: BDSO Server ReliableBroadcast Function

The file comprises two primary functions, ReliableBroadcast and HandleReliableBroadcast, alongside some utility functions that won't be discussed in detail. ReliableBroadcast serves as the initiator of reliable broadcast by appending the "BRACHA_BROADCAST_INIT" tag to the message and transmitting it to all the server's peers. Thereafter, the processing takes place within the HandleReliableBroadcast function.

Since this algorithm is designed to function asynchronously, it is not possible to implement the Reliable Broadcast operation as a single function. Instead, it requires an initiator function to trigger the process, a handler function to manage the broadcast, and a mechanism for storing the state of each message. We have now seen the initiator and handler, therefore let us now provide our way of storing a message state.

Each server needs to know its state towards each message, and the same information for each of its peers. This information is whether the server reached the ECHO or VOTE state. Figure 4.2 depicts the maps that the server struct uses to store message states.

```
// Used for reliable broadcast
My_init    map[string]bool
My_echo    map[string]bool
My_vote    map[string]bool
Peers_echo map[string]bool
Peers_vote map[string]bool
```

Figure 4.24: Reliable Broadcast State Variables

The keys of the map are constructed inside the HandleReliableBroadcast function using the sender id and message content. The key initialization for handling incoming BRB messages is shown in Figure 4.25.

```
my_key := v.Content[1]
peers_key := v.Sender + "{" + v.Content[1] + "}"
```

Figure 4.25: Reliable Broadcast State Variable Key Initialization

Each server's key only requires the message content, whereas the peer keys need to be differentiated by their id. Having these keys in place, the HandleReliableBroadcast function can now keep track of each server's state regarding each message, as shown in Figure 4.26.

```
if v.Tag == BRACHA_BROADCAST_ECHO {
     receiver.Peers_echo[peers_key] = true
}
if v.Tag == BRACHA_BROADCAST_VOTE {
     receiver.Peers_vote[peers_key] = true
}
```

Figure 4.26: Reliable Broadcast State Variable Action

And after counting the messages, the code moves on to changing the current server's state depending on the conditions of Bracha's Reliable Broadcast algorithm. This process is depicted in Figure 4.27.

```
// count messages
echo_count, vote_count := countMessages(receiver, my_key)

// on receiving <echo, v> from n-f distinct parties:
if v.Tag == BRACHA_BROADCAST_ECHO && echo_count >= config.N-
config.F {
          if receiver.My_vote[my_key] {
               v :=
CreateMessageString(BRACHA_BROADCAST_VOTE, v.Content)
               sendToAll(receiver, v)
               receiver.My_vote[my_key] = false
          }
     }
```

Figure 4.27: Reliable Broadcast Condition Checking

When the algorithm finally reaches the desired number of VOTES, it returns true, denoting the message has been successfully transmitted, as shown in Figure 4.28. With this, the reliable_broadcast.go is successfully concluded, returning true.

```
// on receiving <vote, v> from n-f distinct parties:
if v.Tag == BRACHA_BROADCAST_VOTE && vote_count >= config.N-
config.F {
          return true
     }
```

Figure 4.28: Reliable Broadcast Completion Check

Within the messaging package, we also have a file named byzantine_handlers.go. As the name suggests, this file performs similar functions to handlers.go, but is designed specifically for Byzantine servers. This file is instrumental in evaluating the overall system.

```
func handleGetByzantine(receiver *server.Server, message
Message, half_and_half bool) {…}

func handleAddByzantine(receiver *server.Server, message
Message, half_and_half bool) {…}

func handleRBByzantine(receiver *server.Server, message
Message, half_and_half bool) {…}
```

Figure 4.29: Declaration of Byzantine Handler Functions

Each function operates depending on the type of Byzantine behavior of the server. Details on each Byzantine behavior will be provided in the next chapter. The function declarations of the Byzantine Functions are shown in Figure 4.29.

Modules Package

The modules package comprises multiple behaviors that a server can exhibit. Each file within this package contains a single function that initializes several processes equivalent to the value of the NUM_THREADS variable defined in the aforementioned configuration file, along with the corresponding behavior. Figure 4.30 depicts the main task of the behavior functions.

```
func StartNormal(servers []config.Node, default_port, num_threads int) {
…
	for {
		msg, err := s.Receive_socket.RecvMessage(0)
		messaging.HandleMessage(s, msg)
		}
…
}


func StartMalicious (servers []config.Node, default_port, num_threads
int) {
…
	for {
		msg, err := s.Receive_socket.RecvMessage(0)
		messaging.HandleMessageByzantine(s, msg, "MALICIOUS")
		}
…
}
```

Figure 4.30: Byzantine Functions Main Task

Tools Package

The tools package provides the same functionality as the client tools package. That includes logging and statistics related to the experimental evaluation.

Main

The main.go function is the entry point of the code, and it takes in just one parameter representing the behavior of all the threads that this program will create (blank for normal behavior). It is responsible for parsing the hosts and config file and starting the requested amount of threads. Let us provide two example executions.

```
…/server$ go run main.go


| node0:5555 | Bound tcp://*:5555
| node0:5556 | Bound tcp://*:5556
| node0:5556 | Started with NORMAL behavior
| node0:5555 | Started with NORMAL behavior
```

Figure 4.31: Normal Server Execution Output

```
…/server$ go run main.go mute


| node0:5555 | Bound tcp://*:5555
| node0:5556 | Bound tcp://*:5556
| node0:5556 | Started with MUTE behavior
| node0:5555 | Started with MUTE behavior
```

Figure 4.2: Mute Server Execution Output

Figures 4.31 and 4.32 show the console output after their corresponding run commands.

4.4 Atomic-Adds System

Now that we have examined the implementation of the BFT-Distributed-G-Set System (hereinafter referred to as BDSO), we can move on to the analysis of a system that utilizes BDSO as a fundamental building block to address a real-world problem. As the Atomic-Adds is primarily based on the BDSO, we will only be explaining the parts of the code relevant to the new system.

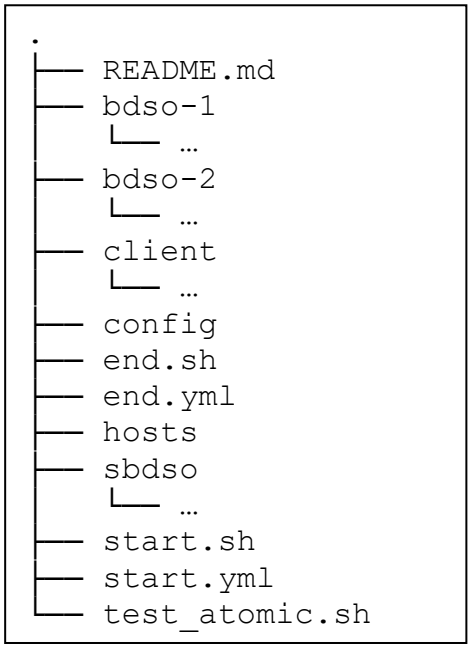

Figure 4.33: 2-Atomic-Adds Code Structure

Similarly, to the BDSO implementation, this project has some scripts for automating the start-up and testing. The project directory stricture is shown in Figure 4.33. The folders bdso-1 and bdso-2 are the same and they contain the BSDO code. The most important part of this project is the sbdso folder, which stands for Smart BDSO. It contains the normal BDSO code, with some additions to support the 2-Atomic-Adds functionality. To understand the code, we first need to explain the topology of the system.

4.4.1 Topology

This system is comprised of four distinct components (see Figure 4.33). Those are the two BDSO networks, one Smart BDSO network, and the clients. The clients can connect to any of the 3 networks they desire. With a connection to a normal BDSO, the client can *Add* and *GET* as we have seen in previous sections. However, when they connect to the SBDSO, they can add atomically to their desired BDSO, using the Add Atomic command. When two records that fit the atomic requirements appear in the SBDSO, they are added like normal to their desired BDSO.

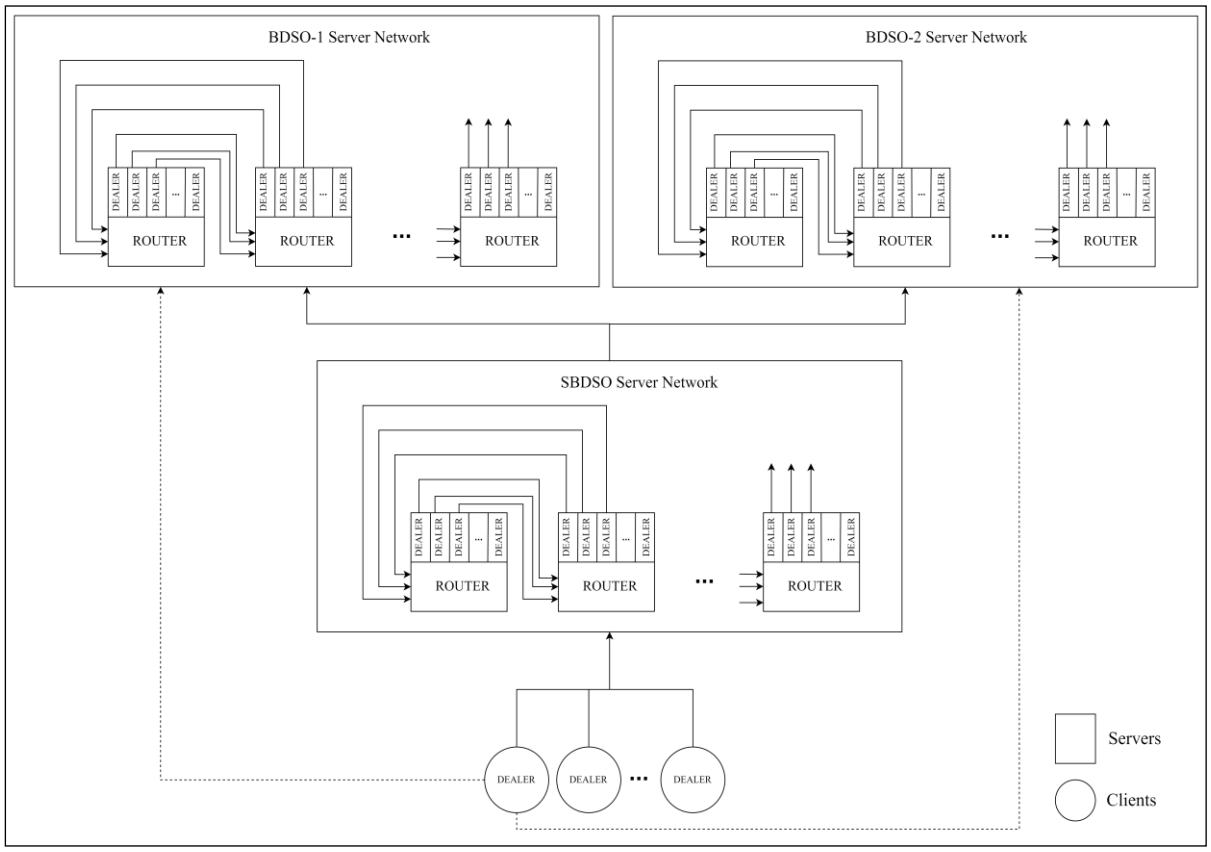

Figure 4.34: 2-Atomic-Adds Topology

4.4.2 Client

The Smart BDSO client has the same functionalities as the BDSO client, with some additional. As we have seen from the topology, the SBDSO client can choose to connect to either one of the BDSOs and perform the normal *ADD* and *GET* operations or connect to the intermediary SBDSO network. This can be done through command line arguments. Example executions can be seen in Figure 4.35.

```
/client$ go run main.go -net sbdso

/client$ go run main.go -net bdso-1
```

Figure 4.35: 2-Atomic-Adds BDSO and SBDSO Execution Examples

When a client connects to the SBDSO with an interactive session, they are prompted differently than previously in the BDSO. They can Add Records Atomically by using the ‘at’ input. When two clients add records that fulfill the necessary atomic qualities, both records are then added to the destination BDSO. Let us explain the example depicted in Figures 4.36 and 4.37.

```
Your ID
> loukas
ID set to 'loukas'

| loukas | Established connection with tcp://localhost:5500
| loukas | Established connection with tcp://localhost:5501
| loukas | Established connection with tcp://localhost:5502
| loukas | Established connection with tcp://localhost:5503
| loukas | Established connection with tcp://localhost:5504
| loukas | Established connection with tcp://localhost:5505
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e' for EXIT
> at
Format of atomic records: peer_id;destination;your_message;peer_message
Record to append atomically > marios;bdso-1;hello;world
| loukas | Called ADD_ATOMIC(0.atomic;loukas;marios;bdso-1;hello;world)
| loukas | Waiting for f+1 ADD_ATOMIC replies
```

Figure 4.36: SBDSO User Interface with Atomic Add Request Example from”loukas”

```
Your ID
> marios
ID set to 'marios'

| marios | Established connection with tcp://localhost:5500
| marios | Established connection with tcp://localhost:5501
| marios | Established connection with tcp://localhost:5502
| marios | Established connection with tcp://localhost:5503
| marios | Established connection with tcp://localhost:5504
| marios | Established connection with tcp://localhost:5505
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e' for EXIT
> at
Format of atomic records: peer_id;destination;your_message;peer_message
Record to append atomically > loukas;bdso-2;world;hello
| marios | Called ADD_ATOMIC(0.atomic;marios;loukas;bdso-2;world;hello)
| marios | Waiting for f+1 ADD_ATOMIC replies
```

Figure 4.37: SBDSO User Interface with Atomic Add Request Example from”marios”

And since the SBDSO also acts as a normal BDSO, when both records are added to the SBDSO we can check using a simple ‘*GET*. These records will be labeled at atomic-complete, to differentiate normal records from those intended for addition in another BDSO.

```
2023/04/19 14:56:45 | marios | Record {loukas;bdso-
2;world;hello} appended to destination
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e'
for EXIT
> g
| marios | Called GET
| marios | Waiting for valid GET_REPLY
| marios | GET completed in: 1.8519ms
| marios | {atomic-complete;loukas;marios;bdso-
1;hello;world} {atomic-complete;marios;loukas;bdso-
2;world;hello}
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e'
for EXIT
>
```

Figure 4.38: GET In SBDSO After Atomic Add Request Example

Figure 4.38 shows how we can connect to BDSO-1 for example, and check if our record has been appended. The user “loukas” wanted to add the record “hello” to “bdso-1”. Therefore, a “*GET*” inside this network should produce the record “hello”, which is shown in Figure 3.39 with green highlighting.

```
Your ID
> loukas
ID set to 'loukas'

| loukas | Established connection with tcp://localhost:5600
| loukas | Established connection with tcp://localhost:5601
| loukas | Established connection with tcp://localhost:5602
| loukas | Established connection with tcp://localhost:5603
| loukas | Established connection with tcp://localhost:5604
| loukas | Established connection with tcp://localhost:5605
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e' for
EXIT
> g
| loukas | Called GET
| loukas | Waiting for valid GET_REPLY
| loukas | GET completed in: 2.1445ms
| loukas | {hello}
Type 'g' for GET, 'a' for ADD, 'at' for ATOMIC-ADD or 'e' for
EXIT
>
```

Figure 4.39: GET In BDSO-1 After Atomic Add Request Example

4.4.3 Smart BDSO

Let us now provide the key parts of the SBDSO implementation. The main idea of the Atomic Adds system is to treat the “atomic” records like normal records but with some added

details. When a user wants to add atomically, they are prompted to provide the id and record of their peer as well as their own and the destination they want the message to end up to. All this information is packed into a single record and added to the G-Set. Figure 4.40 depicts an example of such a record.

```
{atomic;loukas;marios;bdso-1;hello;world}
```

Figure 4.40: Atomic Record Example

Whenever a new record is added, the G-Set checks for atomic records through the `CheckAtomic` function, of which the declaration is shown in Figure 4.41.

```
func CheckAtomic(gset map[string]string) (string, string) {…}
```

Figure 4.41: CheckAtomic Function Declaration

When such records are detected, the handleAtomicAdds function (see Figure 4.42) is called, which has the following responsibilities:

- Breaking down the contents of those records
- Calling the BdsoAdd function which adds the records to their respective destinations.
- Waiting for the 2 BDSOs to reply and then replying to the 2 clients.

```
func handleAtomicAdd(s *server.Server, r1, r2 string) {
	tools.Log(s.Id, "Found atomic records {"+r1+"} with {"+r2+"}")
	var response []string

	// handle
	parts1, parts2 := strings.Split(r1, ";"), strings.Split(r2, ";")
	client1, client2 := parts1[1], parts2[1]
	dest1, dest2 := parts1[3], parts2[3]
	msg1, msg2 := parts1[4], parts2[4]

	// send adds
	BdsoAdd(s, msg1, msg2, dest1, dest2)

	// respond 1
	response = []string{client1, s.Id, ADD_ATOMIC_RESPONSE, r1}
	s.Receive_socket.SendMessage(response)
	tools.Log(s.Id, "Sent ADD_ATOMIC_RESPONSE to "+client1)

	// respond 2
	response = []string{client2, s.Id, ADD_ATOMIC_RESPONSE, r2}
	s.Receive_socket.SendMessage(response)
	tools.Log(s.Id, "Sent ADD_ATOMIC_RESPONSE to "+client2)

}
```

Figure 4.42: SBDSO handleAtomicAdd Function

Chapter 5

Experimental Evaluation

5.1 Experimental Environment

For our experimental evaluation, we used a truly distributed setting using the Cloudlab [23] platform, which allowed us to run our code on several, physically separated machines inside a Local Area Network.

5.1.1 Experimental Scenarios

Before we get into the methodology and experiment structure, it is important to understand the fundamental attack scenarios we tested our system on. Let us present each scenario with a brief description of each one’s behavior.

Normal Scenario

The first and most important execution scenario is one where no Byzantine nodes are present, allowing us to gather failure-free results, which will later act as the basis for the comparison with the rest of the scenarios.

Mute Scenario

In the Mute scenario, a subset of servers equivalent to *f* are not actively engaged in any message exchange but instead serve only as passive recipients of messages. This situation provides an excellent opportunity to showcase the effectiveness of the algorithm in terms of its correctness, specifically when *f* servers remain inactive and do not play a role in the Reliable Broadcast process.

Malicious Scenario

During the Malicious Scenario, *f* servers from the system act maliciously, meaning they have intentions of confusing the system. They do this by changing the messages they receive during relays to the same wrong value. In our case we used "`BYZANTINE_0`". This is the perfect scenario to test the algorithm's ability to withstand malicious actors in the system given that they are at most $f=(n-1)/3$, where n is the total amount of servers in the system.

Half and Half Scenario

The Half and Half attack scenario is a variation of the Malicious Scenario where malicious actors change the contents of the messages they relay. However, in this case, half of the servers receive one message, while the other half receive a different message. The server nodes with an even id number received "`BYZANTINE_0`" and the rest "`BYZANTINE_1`". This scenario aims to test the algorithm's equivoque properties and evaluate its performance under non-standard conditions. By simulating this scenario, it is possible to assess how well the algorithm handles situations where messages are inconsistent across nodes and verify that it can still produce accurate results.

5.1.2 Experiment Structure and Methodology

During the experimental evaluation, we tested various scenarios and configurations which eventually resembled a hierarchy of scenarios. Consequently, we will now define some key terms we will be using to describe this hierarchy.

- **Request:** One client request is equal to one Add operation, followed by one Get operation.
- **Majority Server:** A configuration where the servers outnumber the clients, specifically, 4 clients and 25 servers, 8 of which can be Byzantine.
- **Majority Client:** A configuration where the clients outnumber the servers, specifically, 19 clients and 10 servers, 3 of which can be Byzantine. The reason for the high number of servers when compared to the client number in the Majority Server scenario is that a low number of servers would not allow enough Byzantine nodes to provide any insight when testing such scenarios. As resources were limited, it was difficult to create a scenario with many clients and many servers.
- **Balanced:** To achieve a balanced configuration, we aimed for a nearly equal number of clients and servers, consisting of 14 clients and 15 servers. One machine was designated as the "master," leaving us with 29 remaining machines to distribute equally between the servers and clients.

We divided our experiments into 3 major categories, Majority Server, Majority Client, and Balanced. We then executed the various attack scenarios we mentioned in the previous section, 5 times each. Finally, the results of those 5 iterations were averaged and we had our complete dataset. While we could execute any of these experiments using a differing number of threads, thus changing the number of actors in the system, we opted not to as we believed that would hinder the system’s true distributed nature. Therefore, we will only be using threads to stress test one Majority Server configuration. An indicative graphical representation of the way we collected our results is shown in Figure 5.1.

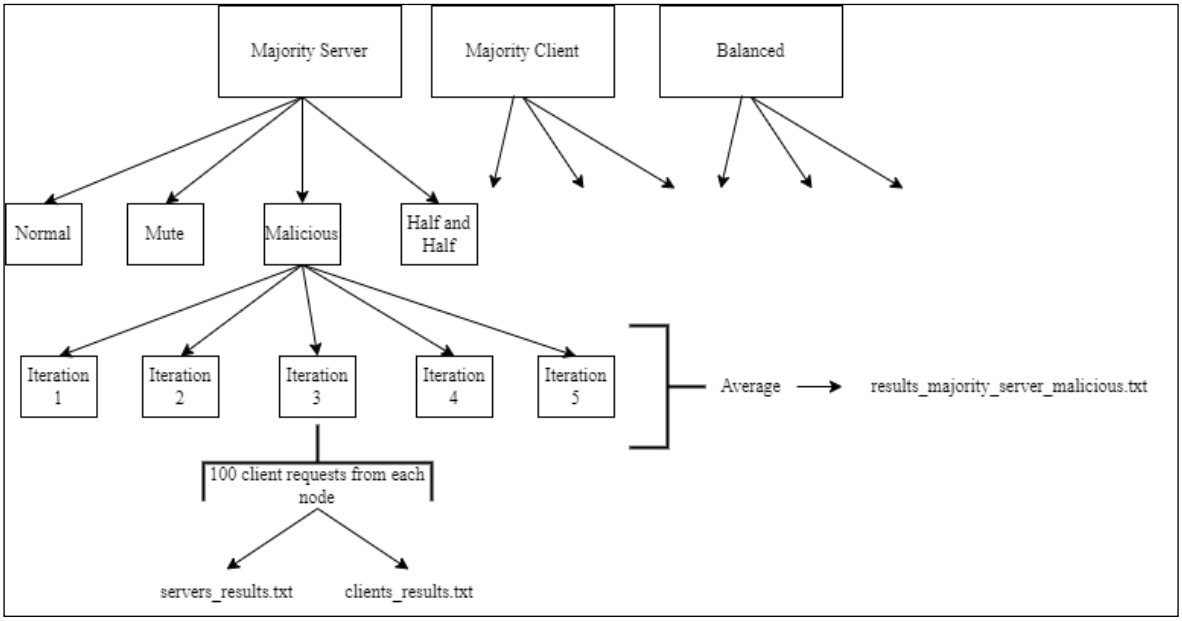

Figure 5.1: Indicative Structure of Experimental Evaluation Results

5.1.3 Machines and Configuration

We ran each actor of the system on a different physical machine. We made sure to use the same type of machines to avoid inconsistent results due to the nodes having different capabilities. We reserved 30 physical machines in the Wisconsin Cluster of CloudLab. Our machines were of type c220g5 [24], see Figure 5.2 for their specifications.

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              40
On-line CPU(s) list: 0-39
Thread(s) per core:  2
Core(s) per socket:  10
Socket(s):           2
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Silver 4114 CPU @
2.20GHz
Stepping:            4
CPU MHz:             801.088
CPU max MHz:         3000.0000
CPU min MHz:         800.0000
```

Figure 5.2: CPU Specs of c22g5

To initialize and control the machines we determined a master node, which was node0. This node's only responsibility was using Ansible [25] and bash scripts to automate the initialization, update, and testing. For example, Figure 5.3 shows a snippet from the start.yml file which starts all machines with a behavior that is based on the hosts file. The absolute paths are not included here for simplicity reasons.

```
- name: Start normal servers
  hosts: servers-normal
  tasks:
    - name: Start server
      raw: cd …/server; nohup go run main.go normal > stdoutfile 2>
stderrfile & sleep 1
```

Figure 5.3: Start Servers Ansible Script Snippet

This part of the Ansible file executes the command "*go run main.go normal*" inside the server directory, for all the hosts that are under the [servers-normal] tag. Therefore, all the nodes under the specified tag, execute this command. However, there is a slight problem. The hosts file is only stored in node0. For this reason, a snippet like the following (Figure 5.4) is required:

```
    - name: Fetch file config from node 0 (master node
      hosts: all
      command: scp loukis@node0~/hosts …/BFT-Distributed-G-Set-Remote
```

Figure 5.4: Copy hosts File from node0 Ansible Script Snippet

Using similar Ansible files and corresponding bash scripts we automated all our processes.

5.1.4 Performance Metrics

To evaluate the overall behavior of the algorithms being studied in our experimental scenarios, it is necessary to measure their performance across various dimensions. As such, every experiment should be evaluated using the following aspects of performance metrics.

- **Average Get Latency:** The average time it takes for a GET request to be fulfilled. It is measured on the client side and the timer begins as soon as the client has sent all the ADD REQUESTS and ends as soon as $f+1$ matching replies are reached.

- **Average Add Latency:** The average time it takes for an ADD request to be fulfilled. It is also measured on the client side and the timer begins as soon as the client has sent all the GET REQUESTS and ends as soon as f+1 have been received for the same request.
- **Average Message Complexity:** The number of messages exchanged inside the system, divided by the total amount of requests, gives us the average number of messages needed for each request to be fulfilled. This metric is measured on the server.
- **Average BRB Message Complexity:** The Average BRB Message Complexity is similar to the Average Message Complexity metric, but it specifically measures the complexity of messages related to the Reliable Broadcast module. By focusing on this metric, we can gain insights into the proportion of messages that the RB module is responsible for. This metric is also measured on the server.
- **BRB to ADD Latency Ratio:** The time taken to broadcast a message expressed as a percentage of the total ADD operation latency. The Reliable Broadcast Latency is measured as the time duration between the first receipt of an RB message and the last. Although this metric is difficult to accurately measure due to the nature of Bracha's Reliable Broadcast Algorithm, we tried to gather as much information as possible. This metric Is taken on the server and can give us an understanding of how much of the ADD time is used for the RB module.

5.2 Evaluation Results of BDSO

In this section, we analyzed the results we have gathered from the experiments. We focused and commented on the most relevant results. Despite that, the entire spreadsheet of results is available in the project's GitHub repository [14].

5.2.1 Configurations and Attack Scenarios Analysis

We first compared the results between the different configurations and attack scenarios that were explained earlier. Let us begin with the operation latencies between the different scenarios and behaviors.

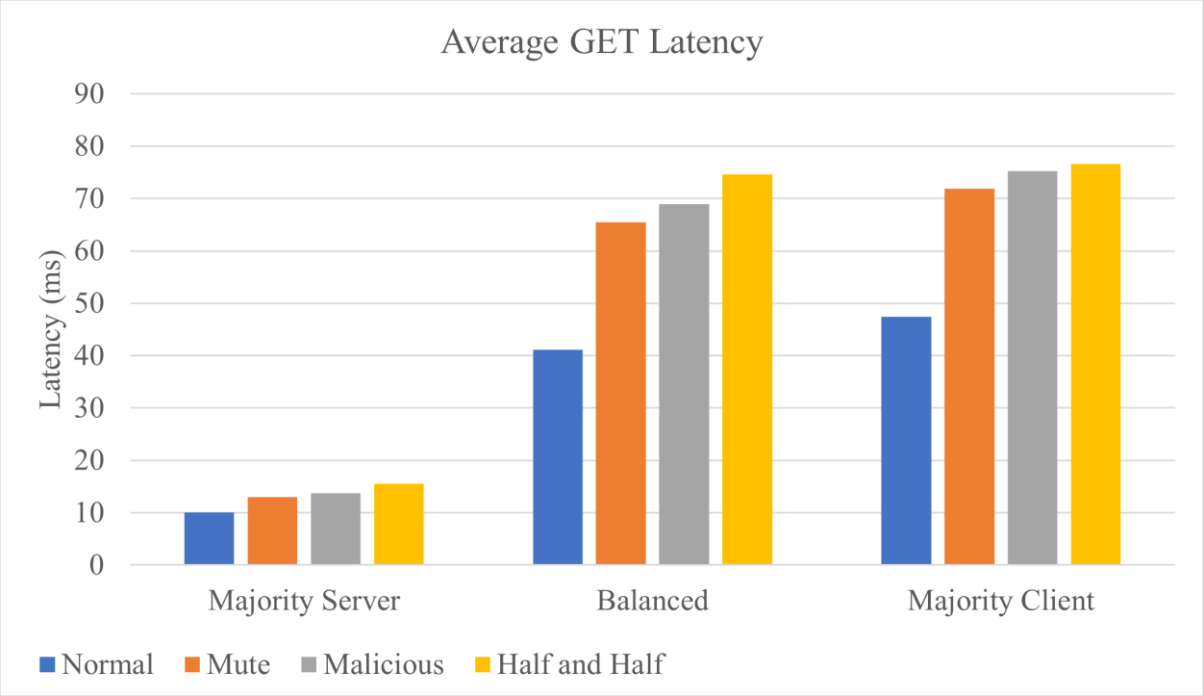

Figure 5.5: Average GET Latency

After analyzing the different configurations based on the average GET time (see Figure 5.5), it was found that the configuration with more clients than servers resulted in the highest operation latency. This outcome was anticipated as an increased number of requests naturally necessitates more time to process. When examining the various attack scenarios, a distinct increase in operation latency is evident when comparing Normal to any attack scenario, primarily in the Balanced and Majority Client configurations. This result was expected as the GET operation relies heavily on client processing, and an influx of irrelevant messages can cause the algorithm to take longer to locate the necessary matching replies. Additionally, the half-and-half behavior exhibited slower response times than other malicious behaviors, by approximately 5-10 milliseconds. Unsurprisingly, the "mute" attack scenario exhibited a slightly faster execution time compared to the other attack scenarios. This is due to the fact that this behavior does not transmit any messages, thus eliminating the unnecessary burden on the clients.

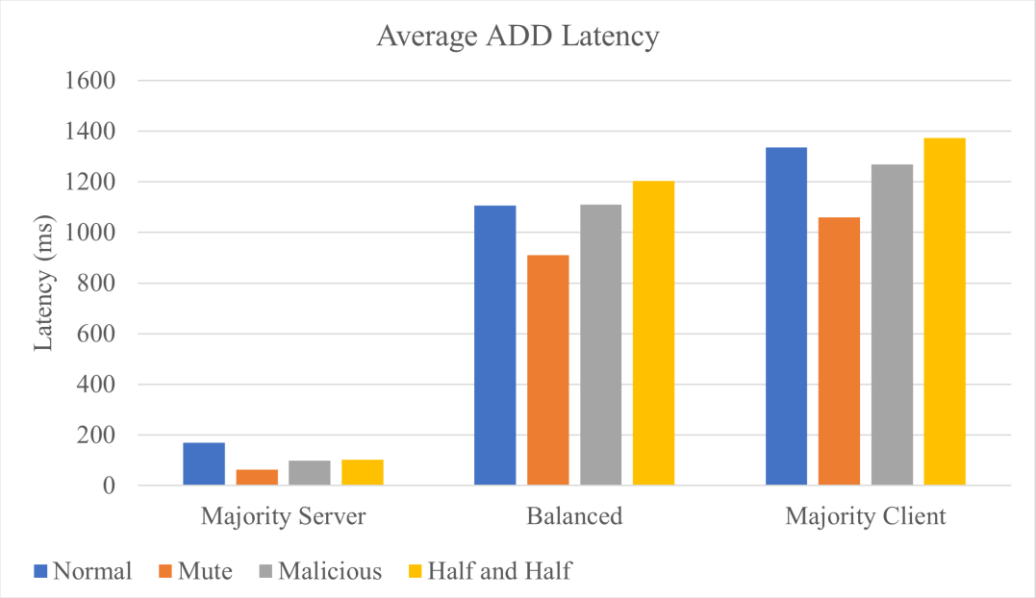

Figure 5.6: Average ADD Latency

Similarly, when examining the ADD (see Figure 5.6) operation latency in terms of configuration, it was found that the majority-client configuration was the slowest, with an average processing time of approximately 1300 milliseconds. Upon analyzing the ADD operation latency concerning the attack scenario, it was observed that Normal and Malicious behaviors exhibited similar performance, while the Half and Half attack scenario resulted in slightly longer processing times. However, the Mute attack scenario was the quickest of all, which can be attributed to the low volume of messages exchanged within the system.

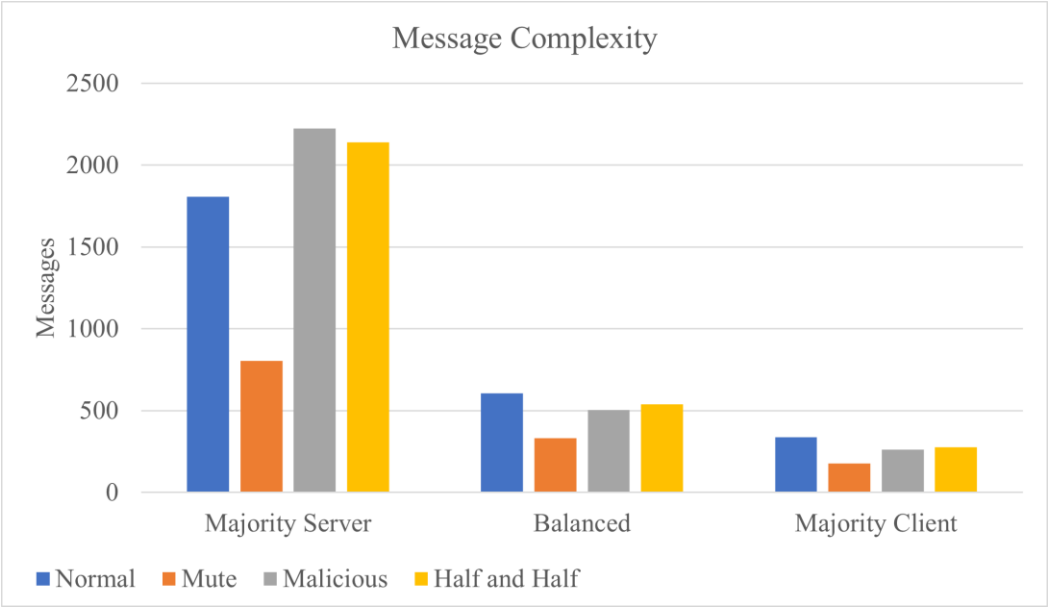

Figure 5.7: Message Complexity

As with the ADD operation latency, it was found that the communication within the system significantly increases when all servers are operational, resulting in a greater message complexity compared to other attack scenarios (see Figure 5.7). Conversely, the other attack scenarios exhibit similar message complexities. When comparing configurations, it is apparent that message complexity increases proportionally with the number of servers in the system. This can be attributed to the increased workload on the reliable broadcast module, which leads to a higher volume of messages exchanged within the system.

Let us now present a graph (Figure 5.8) we believe is crucial to the evaluation of this algorithm. What the following graph shows is the ratio between the time required to complete a Reliable Broadcast, taken from the servers, and the time necessary to complete an ADD operation. In essence, what we see here is how much of the time a client waits for an ADD is spent broadcasting the message.

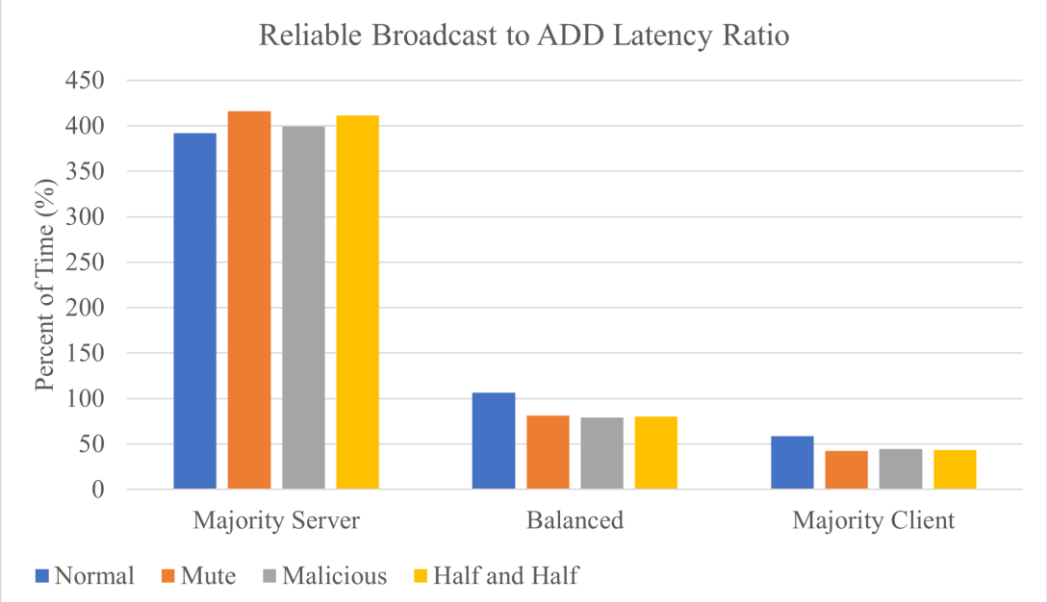

Figure 5.8: Reliable Broadcast to ADD Latency Ratio

While we expected to see percentages just below 100%, we saw numbers up to 400%, in the Majority-Server configuration. Although the other configurations closely matched our prediction, the configuration led us to an interesting finding about the algorithm. As we know from Bracha's Reliable Broadcast Algorithm [10], if there are *n* nodes in the system, each node broadcasts a message to all other nodes, resulting in *n*-1 messages being sent by each node. The nodes then exchange acknowledgments with each other, resulting in another *n*-1 messages per node. The total number of messages exchanged is therefore (*n*-1)^2. This means

that, as the number of operational servers increases, the number of messages exchanged also increases quadratically. However, the clients require f+1 matching replies to conclude the operation. Therefore, as more servers are present in the system, the number of messages clients require grows linearly, while the number of messages exchanged by the servers grows quadratically. The percentages we observe in Figure 5.8 for the Majority-Client configuration suggest that the Reliable Broadcast is still going on, trying to conclude, even after the ADD operation is fulfilled. This is a possible shortcoming of the algorithm.

5.2.2 Performance Analysis

After studying the effects of different configurations on the system, we sought to test its scalability. To achieve this, we increased the number of threads used by each machine, with each thread representing a separate machine. For instance, if a physical machine was acting as a client with three threads, it would send requests to three different clients. Similarly, for servers, each physical machine with a certain behavior would have three separate actors active in the network, exhibiting the same behavior. Regrettably, we could only achieve efficient performance with up to three threads since each server maintains its own G-Set in the runtime memory. Increasing the number of threads caused the master process to run out of memory or significantly increase the operation latency, which we believe would be misleading regarding the actual performance of the system. Three threads in this system mean 75 server processes and 12 client processes. Let us now provide the most relevant results.

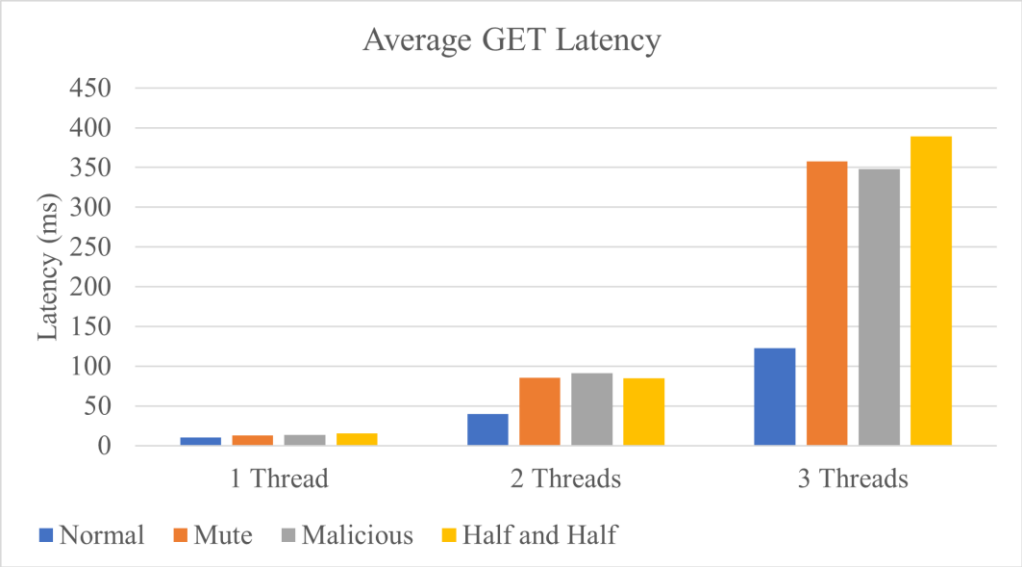

Figure 5.9: Average Add Latency Performance Test

Upon analyzing the GET operation latency (Figure 5.9), it becomes evident that the normal scenario exhibits a lower degree of degradation compared to the attack scenarios. In contrast, the attack scenarios show a significant increase, with the Mute attack scenario showing the least impact and the Half and Half scenario showing the most. This increase was expected since the introduction of each additional thread leads to not only servers but clients too, generating requests, which in turn increases the overall workload of the system.

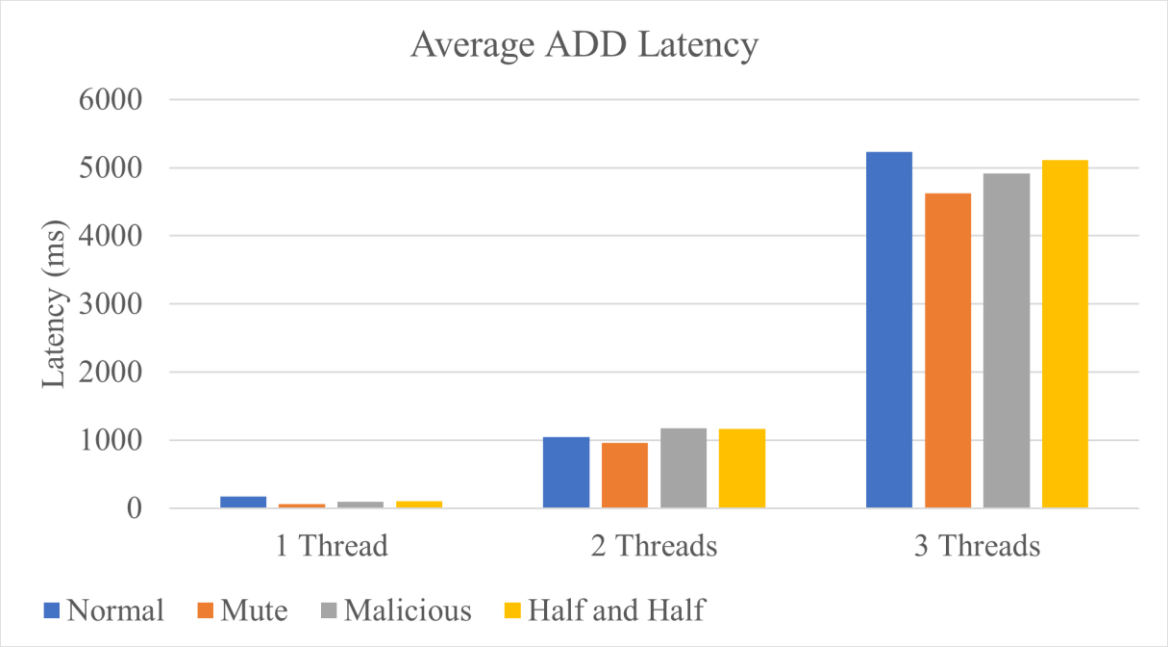

Figure 5.10: Average Add Latency Performance Test

When it comes to the latency of the ADD operation (Figure 5.10), the jump is even more significant, going from around 100 milliseconds to around 5 seconds. These results are not surprising as we expected a significant decrease in performance due to the introduction of additional threads, servers, and clients. However, these results suggest that the system does not efficiently scale, which can be attributed to the large number of messages exchanged within the system.

As we have seen, communication between the servers is extremely burdensome. Let us directly illustrate this using Figure 5.11. For a system with just 75 servers and 12 clients, the network changed over 90 million messages.

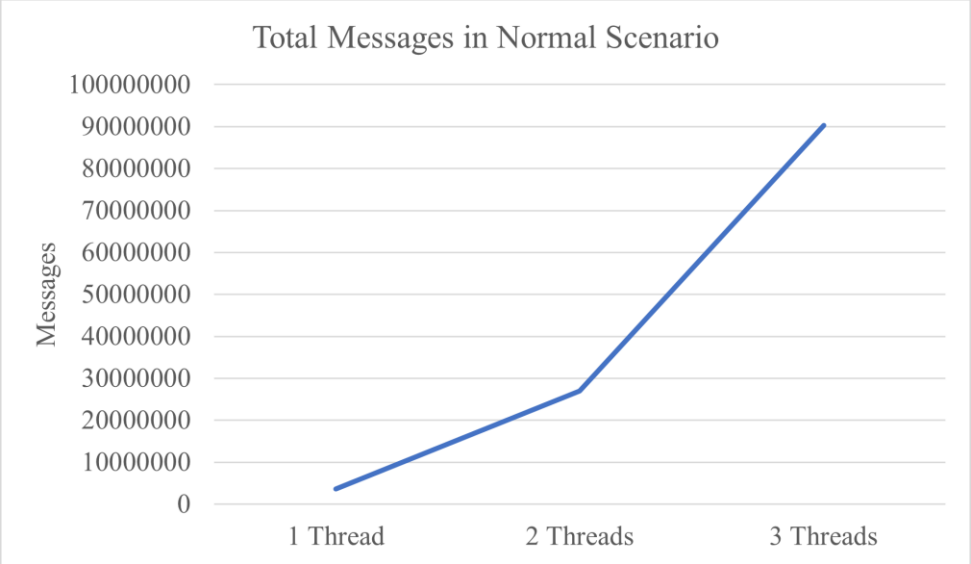

Figure 5.11: Total Number of Messages Exchanged

5.3 General Evaluation of SBDSO

The evaluation of the SBDSO system was not as extensive as that of the BDSO system, as we had already tested its primary components earlier. As a reminder, the SBDSO system works by allowing a client to instruct the SBDSO to add a record to a BDSO. When the SBDSO receives two atomic messages that correspond to each other, it adds them to their respective BDSOs. Our analysis focused on the latency experienced by clients, and the latency observed by the SBDSO when adding a record to a BDSO.

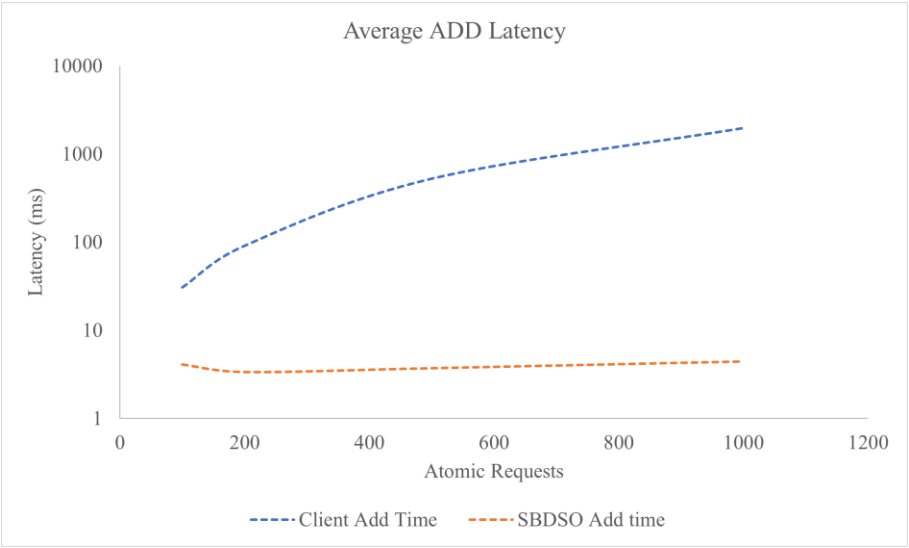

Figure 5.12: Comparison of Client and SBDSO Add Time for Increasing Atomic Requests

From our analysis, it is evident that the operation latency observed by the SBDSO remains constant regardless of the number of requests. However, the operation latency observed by clients increases linearly on a logarithmic scale, resulting in quadratic growth. This suggests that most of the time that a client observes is spent waiting for the SBDSO to locate the necessary atomic records before adding them to their respective BDSOs. This finding aligns with expectations, as this step is a crucial part of the atomic add process.

5.4 Experimental Summary

Our experiment provided us with valuable insight into the limitations of consensus-free algorithms, like the one we examined. While these algorithms can be efficient and innovative, they come with a significant cost, which is a large volume of messages. As we observed, the number of messages grows quadratically, particularly those related to the Reliable Broadcast, and this underscores the importance of minimizing the total number of servers to reduce message complexity. It also became apparent from the performance testing that the system does not scale well as the operation latency does not grow proportional to the number of machines in the system, but rather quadratically. We thus concluded that the Reliable Broadcast algorithm used in the system is the main bottleneck, leading to a significant increase in message complexity. However, this degradation of performance was expected since in order to achieve consistency in the presence of Byzantine-Faults and especially asynchronously comes at big a cost. That being the large number of messages exchanged in the system. Bearing that in mind, there are some ways in which the strong guarantees could be loosened up thus making the algorithms less demanding. While the system is currently designed to operate in a fully asynchronous manner, it is important to acknowledge that in commercial usage, complete asynchronicity might not be necessary. Hence, there is room for optimization by considering synchronization issues to reduce the overall number of messages exchanged within the system. By addressing these synchronization challenges, we can strike a balance between efficiency and real-world commercial requirements, resulting in a more streamlined and effective system.

Chapter 6

Conclusion

6.1 Summary

Throughout this thesis, our primary objective was to implement and assess a novel type of distributed ledger system known as the Distributed G-Set, as introduced in [4]. This pursuit stemmed from the escalating significance of blockchain technology and the requirement for resilient fault tolerance mechanisms to safeguard the stability and integrity of these systems. After an extensive study of relevant literature, we proceeded with implementing this system utilizing the Go programming language, alongside the ZeroMQ messaging library. We then performed multiple tests on a genuinely distributed network of machines within a LAN. Upon analyzing the outcomes, we have identified inherent shortcomings in the algorithms that arise as a trade-off for the level of consistency they guarantee.

6.2 Challenges

Undoubtedly, one of the most challenging aspects of this thesis was acquiring proficiency in the relevant technologies while utilizing them. While we found the Go programming language relatively easy to learn, the same cannot be said for ZeroMQ. Mastering the usage of ZeroMQ sockets and comprehending the distinctions between the various messaging patterns proved to be quite challenging. Besides the primary technologies, we also had to

employ other tools such as Bash programming and Ansible scripting. These were mainly utilized to control the various machines from a single node. That brings us to the management of multiple machines. Debugging code and ensuring its proper functioning on a single machine can be difficult enough, but the task becomes even more complex when dealing with over 30 machines. The last significant challenge we encountered was achieving true asynchrony in the implementation. The algorithms were described in pseudocode and declarative language, which did not provide insights into how to achieve asynchrony. Therefore, we had to test various approaches, with many of them not functioning properly until we found one that satisfied our requirements.

6.3 Future Work

With the conclusion of this thesis, we would like to list some suggestions if the work of this thesis is to ever be expanded upon. Firstly, the codebase is far from production-ready or anything similar. We were developing the system while learning the technologies used, therefore, best practices and perfect code organization were not always the case. An overhaul of the codebase is necessary, with a focus on organization and performance optimizations, having in mind the best practices of Go, ZeroMQ, and Distributed System Design. Another step in expanding the work of this thesis is to evaluate its performance using another broadcast algorithm or an optimized version, since this was the main bottleneck of our implementation. Success in producing better results means that the system with the right broadcast algorithm could be industrially viable. The final and perhaps most exciting expansion of the Byzantine Fault-Tolerant Distributed G-Set is the development of a system capable of tracking ownership of assets, and their exchange using the 2-Atomic-Adds system that was proposed earlier. A system capable of tolerating one of the most demanding types of faults in distributed computing has truly limitless real-world applications, and this is what this thesis sought to demonstrate.

Bibliography

[1] Antonio Fernández Anta, Chryssis Georgiou, Kishori Konwar, and Nicolas Nicolaou, Formalizing and Implementing Distributed Ledger Objects, ACM SIGACT News, Volume 49, Issue 2, pp. 58-76, June 2018.

[2] Antonio Fernández Anta, Chryssis Georgiou, and Nicolas Nicolaou, Atomic Appends: Selling Cars and Coordinating Armies with Multiple Distributed Ledgers, Proc. of International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019), pp. 5:1-5:16, Paris, France, 2019.

[3] Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, and Michel Raynal, Atomic Appends in Asynchronous Byzantine Distributed Ledgers, Proc. of the 16th European Dependable Computing Conference (EDCC 2020), pp. 77-84, Munich, Germany, Online, 2020.

[4] Vicente Cholvi Juan, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, Michel Raynal, and Antonio Russo, Byzantine-tolerant Distributed Grow-only Sets: Specification and Applications, Proc. of the 4th International Symposium on Foundations and Applications of Blockchains (FAB 2021), pp. 2:1-2:19, Davis, CA, USA, Online, 2021.

[5] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Retrieved April 20, 2023, from https://bitcoin.org/bitcoin.pdf

[6] Buterin, V. (2013). Ethereum whitepaper. Ethereum. https://ethereum.org/en/whitepaper/, (Accessed: 2023-04-20)

[7] Solana. (2017). Solana: A new architecture for a high performance blockchain. https://solana.com/solana-whitepaper.pdf, (Accessed: 2023-04-20)

[8] Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, 4(3), 382-401. https://doi.org/10.1145/357172.357176, (Accessed: 2023-04-20)

[9] Castro, M., & Liskov, B. (1999). Practical Byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI) (Vol. 99, pp. 173-186), (Accessed: 2023-04-20)

[10] Gabriel Bracha. Asynchronous Byzantine Agreement Protocols, Inf. Comput., 75(2):130-143, (1987)

[11] Abraham. (2020, September 19). Living with Asynchrony: Bracha's Reliable Broadcast. decentralizedthoughts.github.io. Retrieved April 20, 2023, from https://decentralizedthoughts.github.io/2020-09-19-living-with-asynchrony-brachas-reliable-broadcast/

[12] Redman. (2020, August 2). Triple-Entry Bookkeeping: How Satoshi Nakamoto Solved the Byzantine Generals' Problem, news.bitcoin.com. Retrieved April 20, 2023, from https://news.bitcoin.com/triple-entry-bookkeeping-how-satoshi-nakamoto-solved-the-byzantine-generals-problem/.

[13] Vasilis Petrou. Implementation and Evaluation of a Randomized Byzantine Fault Tolerant Distributed Algorithm, Diploma Project, Dept. of Computer Science, University of Cyprus, (2021)

[14] Implementation And Experimental Evaluation Of Byzantine-Tolerant Distributed Sets. https://github.com/loukaspapalazarou/Thesis (Accessed: 2023-04-20)

[15] The Go Programming Language. https://golang.org, (Accessed: 2023-04-20)

[16] Using Go Modules, https://go.dev/blog/using-go-modules, (Accessed: 2023-04-20)

[17] Go maps in action, https://go.dev/blog/maps, (Accessed: 2023-04-20)

[18] Hintjens, P. (2013, March 12). ZeroMQ: Messaging for Many Applications.

[19] Gauravaram, P., & Knudsen, L. R. (2003). Cryptanalysis of SHA-512. In International Conference on Cryptology in India (pp. 1-12). Springer.

[20] ZeroMQ. https://zeromq.org/, (Accessed: 2023-04-20)

[21] ZeroMQ Socket API. https://zeromq.org/socket-api/, (Accessed: 2023-04-20)

[22] Chapter 3 - Advanced Request-Reply Patterns, https://zguide.zeromq.org/docs/chapter3/#Worked-Example-Inter-Broker-Routing, (Accessed: 2023-04-20)

[23] The CloudLab Manual, https://docs.cloudlab.us, (Accessed: 2023-04-20)

[24] Type info for c220g5, https://www.wisc.cloudlab.us/portal/show-nodetype.php?type=c220g5, (Accessed: 2023-04-20)

[25] How Ansible Works, https://www.ansible.com/overview/how-ansible-works, (Accessed: 2023-04-20)

[26] Canva. (2022, January 3). Free Media License Agreement. Canva. https://www.canva.com/policies/free-media-license-agreement-2022-01-03/, (Accessed: 2023-04-20)

[27] Diagrams.net. (n.d.). Usage terms for diagrams created in diagrams.net. https://www.diagrams.net/doc/faq/usage-terms, (Accessed: 2023-04-20)

[28] Kshetri, N. (2019). Distributed ledger technology in supply chains: A comprehensive literature review and framework for future research. In M. Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology (4th ed., pp. 3956-3965). IGI Global. https://doi.org/10.4018/978-1-5225-9639-4.ch318

[29] IEEE Xplore. (2018). Fault Tolerance in Distributed Systems: A Survey. doi: 10.1109/ACCESS.2018.2887879

[30] Vogels, W. Eventually consistent. Commun. ACM 52, 1 (2009), 40–44