Individual Diploma Thesis

IMPLEMENTATION AND EVALUATION OF A RANDOMIZED BYZANTINE FAULT TOLERANT DISTRIBUTED ALGORITHM

Vasilis Petrou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2021

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

Implementation and Evaluation of a Randomized Byzantine Fault Tolerant Distributed Algorithm

Vasilis Petrou

Supervisor: Chryssis Georgiou

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

May 2021

Acknowledgements

For the completion of this thesis, I would like to express firstly and mostly, my sincere gratitude to my supervisor Associate Professor Chryssis Georgiou, who gave me the opportunity to work on this project and also the excellent collaboration we had throughout the semesters. During the year he helped me towards achieving the best result I could. Additionally, I would like to give my gratitude to the Postdoctoral Researcher Ioannis Marcoullis for providing advice during the meetings and helping me a lot on the further understanding of the algorithms.

Furthermore, I would like to thank the Computer Science Department of the University of Cyprus for the courses and the knowledge provided that helped me during the implementation of my thesis. Finally, I would like to thank my family and friends for being with me during my life and supporting me at every step.

Abstract

Byzantine Fault Tolerant (BFT) algorithms have been under research for many years now, as they provide system protection against node failures or malicious attacks. This document presents an implementation, in the Go programming language with the ZeroMQ messaging library, of the asynchronous randomized BFT algorithm, as presented by Correa et al. in 2005.

This algorithm proposes a stack of three Byzantine-resistant protocols: Multi-valued Consensus, Vector Consensus and Atomic Broadcast. The protocols share a set of chief structural properties; they do not use digital signatures, they make no synchrony assumptions, they are completely decentralized, and they have optimal resilience as they tolerate the failure of $f=(n-1)/3$ nodes, where n is the number of nodes (servers) running the algorithm.

This thesis aims to experimentally evaluate the algorithm on a cluster of machines, in an attempt to better understand the algorithm's performance beyond its theoretical analysis. Furthermore, we aim in comparing its performance with other BFT algorithms. In general, the results touch the surface of the theoretical performance evaluation, with operation latency being around f seconds, when the machines running as servers have good enough resources, and message complexity at approximately fn^3 messages.

Contents

List of Figures

Notable Abbreviations

- ➔ **ABA**: Asynchronous Binary Agreement
- ➔ **ABC**: Atomic Broadcast
- ➔ **ACK**: Acknowledge (message type in computer networking)
- ➔ **ACS**: Asynchronous Common Subset
- ➔ **AVID**: Asynchronous Verifiable Information Dispersal Protocol
- ➔ **B**: batch size
- ➔ **BC**: Binary Consensus
- ➔ **BFT**: Byzantine Fault Tolerance
- ➔ **BVB**: Binary-value broadcast
- ➔ **CSP**: Communicating Sequential Processes
- ➔ **f**: number of Byzantine nodes in a system
- ➔ **FLP Impossibility Result**: Fischer-Lynch-Patterson Impossibility Result
- ➔ **HB**: Honey Badger Protocol
- ➔ **MVC**: Multi-valued Consensus
- ➔ **n**: total number of processes in a system
- ➔ **OPP**: Object-Oriented Programming
- ➔ **PBFT**: Practical Byzantine Fault Tolerance
- ➔ **PK**: Public Key
- ➔ **RBC**: Reliable Broadcast
- ➔ **SK**: Secret Key
- ➔ **SMR**: State Machine Replication
- ➔ **TPKE**: Threshold (Public Key) Encryption
- ➔ **tps**: Transactions per second
- ➔ **VC**: Vector Consensus

Chapter 1

Introduction

1.1. Motivation

The *consensus problem* is a heavily studied problem in distributed computing [7, 8, 9] with both theoretical and practical interests. It is really important for critical applications that rely on the agreement between different processes to execute a task or reach a decision. Examples of such applications include, a transaction commit in a distributed database system, where consensus must ensure consistency, or leader election algorithms, where a system component is designated to initiate and conduct a task.

The term consensus [6, 12] was introduced to describe the agreement on a value among a set of n different processes, where up to f might exhibit arbitrary (Byzantine) behaviour. Each node holds a private value and all non-faulty nodes must agree on one of those values.

The FLP impossibility result [10], a fundamental principle in distributed computing, states that a deterministic algorithm is not capable of solving the consensus problem in an asynchronous system, even with the presence of just a single crash fault. Prior work has introduced a number of different approaches to circumvent the FLP impossibility result, however, most of these approaches make certain synchrony assumptions that are unlikely to hold in real distributed systems, which are usually regarded to be asynchronous.

Therefore, a new approach was introduced to solve the consensus problem in purely asynchronous systems: *randomization* [1, 18, 19, 20, 21]. Randomization helps to ensure the liveness property in a protocol, that is, the protocol will eventually terminate, thus

circumventing FLP. Algorithms adopting this approach are called *probabilistic* or *randomized* and guarantee to solve consensus, by sacrificing determinism. This means that the algorithm adds a form of randomization in some cases like a coin flip, or what is called in bibliography Common-Coin algorithm [22]. This allows the asynchronous randomized BFT algorithms' properties to hold with high probability, without assuming any form of synchrony.

The last few years, more and more randomized algorithms are being published with the aim of becoming more efficient than the previous ones, or even extending the functionality to be more versatile, that is to work differently based on the system that is going to be used for. In these modern times we live in, cloud systems are widely used all over the world via the Internet, implicating that lots of other applications are critically dependent on the correct operation of such systems. Thus, continuous correct operation of cloud systems is absolutely necessary, otherwise the consequences are going to be severe, with the key issues being *reliability, security, performance and scalability* [6].

On that account, when focusing on the reliability issue, it is here that the importance of Byzantine Fault Tolerant (BFT) algorithms [6] emerges. These algorithms will ensure that users' data are correctly and most importantly in the same order, stored in each one of the distributed servers, for instance.

Many algorithms have been designed to solve the BFT problem, from partially synchronous, like the one in [17], to randomized asynchronous algorithms that use threshold cryptography, such as the ones in [18, 19, 21], to asynchronous algorithms that avoid the large overhead of cryptography as the one in [1], to even newer algorithms that attempt to reduce the message complexity using a whole different approach, like the one in [20].

All these algorithms have the common objective of making distributed systems reliable against Byzantine processes. Consequently, we can all agree that the theoretical study and practical development of such algorithms is vital in our current society and with the evolution we can achieve, the future seems much brighter.

1.2. Objective and Contribution

The main objective of this work is to implement and evaluate the Byzantine Fault Tolerant Randomized algorithm of Correa et al [1]. Specifically, we aimed in researching, studying and implementing the algorithm, executing it in real-world-like simulations, validating whether the algorithm's purpose is achieved, and comparing its performance against other algorithms that serve the same goal. Additionally, we focused on discussing its applicability in real applications, by taking numerous measurements such as operation latency, message complexity and size, and determining the difficulty of implementing it. The final objective was to match up these metrics with the theoretical evaluation presented in [1], to check whether they really apply.

1.3. Methodology

Initially, we began by reading several, more general material around Fault Tolerance. We firstly read some basic literature on distributed systems [7, 8, 9], state machine replication [11], consensus [5, 12], and Byzantine Fault Tolerance [6]. Afterwards, we studied one of the fundamental BFT algorithms, the PBFT algorithm of [17], which is neither asynchronous nor randomized, but weak synchronous. However, it gave us the basic understanding regarding the subject. Then, we focused on investigating various asynchronous randomized BFT algorithms, with these algorithms having similar modules and structures, but at the same time being different from each other regarding efficiency approaches. From the five (5) algorithms we researched [1, 18, 19, 20, 21], we chose the mentioned above one to implement. Next, we learned and practiced on the Go programming language [13] as well as the ZeroMQ messaging library [14], by developing several small projects in order to get familiar with these two technologies. Also, we studied the implementation of an older thesis [23], to familiarize ourselves with a similar development approach that used the Go programming language with the ZeroMQ framework to build a Self-Stabilizing Byzantine Fault Tolerance algorithm.

During the course of the implementation, a form of an Agile Software Development process took place, with sprints of one (1) or two (2) weeks and various changes depending on the challenges appearing during the process. After the implementation and

the testing, we ran it locally on a single machine, for debugging and validating the correct operation of the algorithm; we tested its performance and behaviour in a real-world distributed environment, a cluster of nine (9) machines. Finally, we took several execution measurements and compared them with the theoretical evaluations of [1] and also, other BFT algorithms, like the Hashgraph protocol [20] and BEAT [19], to evaluate the dynamic of the algorithm against similar ones that attempt to solve this problem.

1.4. Document Organization

In Chapter 2, we make a background retrospective on Fault Tolerance and Consensus, the ZeroMQ messaging library, the Go programming language and illustrate some related work on BFT algorithms. In Chapter 3, we give a thorough explanation of the randomized BFT algorithm we implemented and present in detail its modules. Next, in Chapter 4, we provide a detailed description of our implementation and the factors behind our main decisions. In Chapter 5, we portray our experimental environment and our experimental scenarios, which we designed to confirm the functionality of our implementation, and we discuss the results we have obtained. Finally, in Chapter 6, we derive our final conclusions and discuss the challenges we faced.

Chapter 2

Background and Related Work

2.1. Fault Tolerance

2.1.1. The Byzantine Generals Problem

Byzantine Fault Tolerance (BFT) originates from the Byzantine Generals Problem [5]; the hypothetical scenario in which the army of the Byzantine empire has multiple separate divisions surrounding an enemy city, and they have to all agree on a common plan to attack or retreat. All, or at least nearly all, must perform the same action or else they will risk complete failure. The problem is that the generals of each division can communicate

with each other via a messenger, and there may be traitorous generals who send different information to different generals. Such examples are illustrated in Figure 2.1.

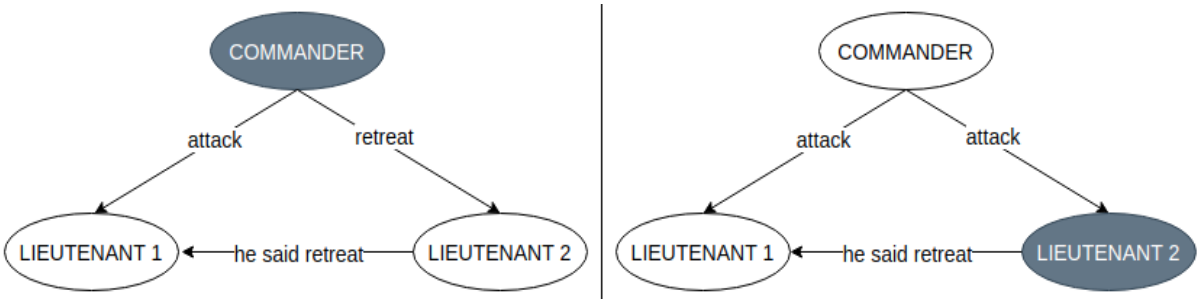

Figure 2.1: Byzantine Generals Problem [5]

In [5], it was proven that the Byzantine Generals Problem has no solution, and thus consensus, when at least one-third of the generals are traitors. Hence, BFT solutions assume that up to *f* nodes out of *3f+1* in total can be Byzantine (malicious).

2.1.2. Correctness Properties of Distributed Systems

BFT applies in distributed systems, so let us take a step back and look at the correctness properties of distributed systems, which can be described in terms of *liveness* and *safety* [7]. Both properties must be satisfied for the system to achieve a strict and ideal sense of correctness.

A *liveness* property is a guarantee that something good will eventually happen with examples being a guarantee that a computation will eventually terminate or that the customer will eventually be served. In a system where the goal is consensus, a liveness property would be a guarantee that each component will eventually decide on a value (termination).

A *safety* property is a guarantee that something bad will never happen, for instance a guarantee that a system will never deadlock or that an innocent person will never be convicted of a crime. In consensus, a safety property would be a guarantee that different components will never decide on different values (agreement).

2.1.3. FLP Impossibility Result

After the correctness properties of distributed systems we have the FLP impossibility [10], which states that both liveness and safety cannot be satisfied in an asynchronous

distributed system, as shown in Figure 2.2, if it is to be resilient to at least one fault. Consequently, FLP impossibility states that asynchronous fault tolerant systems cannot simultaneously satisfy liveness and safety, thus, fault tolerance and correctness cannot be achieved.

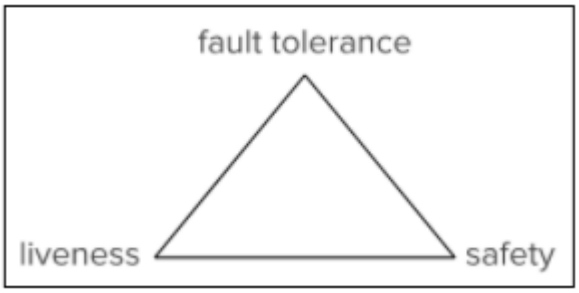

Figure 2.2: FLP Impossibility - both properties cannot be achieved simultaneously

In asynchronous systems, there is no upper bound on the amount of time processes may take to receive messages. So, something needs to be introduced to help overcome this obstacle. First approach is to assume a weak form of synchrony in order to achieve both liveness and safety. The other, more recent and preferable approach is to introduce a form of randomization in the algorithm to either liveness or safety property, using for example coin flip algorithms [22].

2.1.4. Consensus

Consensus simply means a general agreement, thus in a system it requires servers to agree on some value. In a decentralized system, achieving consensus is one of the most important and most difficult tasks. For a network to function correctly, the majority of processes must agree on what is true, reaching consensus at regular intervals. The problem, as it has been already mentioned, is that some nodes may behave arbitrarily or even fail, so the system must be designed in such a way that deals with this inevitability.
Therefore, consensus must satisfy three (3) properties [9]:

i. Termination which states that eventually every non-faulty node decides on a value.
ii. Integrity which supports that if a non-faulty node decides on a value *v*, then *v* must have been proposed by some non-faulty node.
iii. Agreement expressess that every non-faulty node must agree on the same value.

2.1.5. Byzantine Fault Tolerance

Byzantine Fault Tolerance (BFT) [6] refers to the characteristic of distributed systems that allows them to reach consensus in the face of Byzantine faults [5], basically in situations where processes of the system will fail or act arbitrarily and often present conflicting information to different nodes in the system. Therefore, the goal of BFT systems is to be able to resist up to one-third (*1/3*) of the nodes acting maliciously and continue functioning correctly as long as the other two-thirds (*2/3*) of the network reaches consensus.
For instance, all decentralized blockchains [12] run on consensus protocols that all nodes in the blockchain must follow in order to participate, such as Proof-of-Work and Proof-of-Stake that are Byzantine Fault Tolerant and are thus able to resist up to one-third (*1/3*) of the nodes disagreeing.

2.2. BFT Algorithms

2.2.1. Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) [17] is one of the first efficient Byzantine fault tolerant algorithms [6] that claims to be working in asynchronous environments, but in reality assumes weak synchrony to achieve the liveness property, which ensures that clients will eventually receive replies to their requests. Additionally, the safety property states that all non-faulty replicas agree on a total order for the execution of requests despite failures. The algorithm can tolerate up to $f=(n-1)/3$ Byzantine nodes, it is based on the state machine replication method and uses cryptographic algorithms such as signatures to ensure that everything stays unforgeable.
The algorithm is roughly based on four (4) steps; the client sends a request to the primary node, which multicasts the request to the replicas, and then replicas execute the PBFT algorithm for the request and send a reply to the client, who waits for $f+1$ replies from different replicas. Basically, PBFT consensus consists of three phases: Pre-Prepare, Prepare, and Commit, like it can be seen in Figure 2.3.

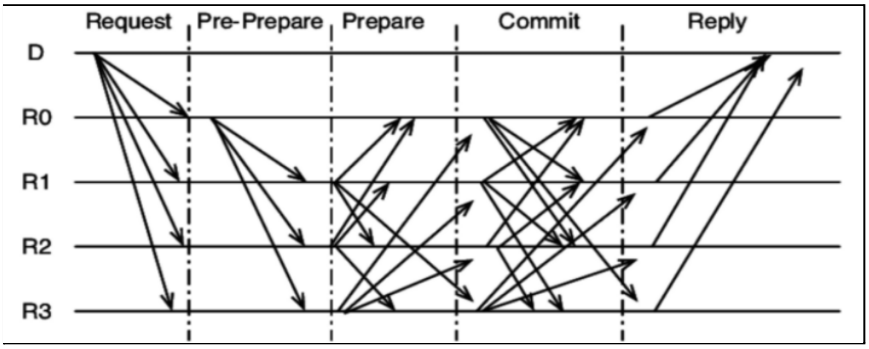

Figure 2.3: PBFT consensus reaching process [17]

2.2.2. The Honey Badger of BFT Protocols

Honey Badger BFT (HB) claims to be the first practical asynchronous BFT protocol [18], in comparison with PBFT, but to overcome the FLP impossibility result, randomization is introduced and thus properties hold with high probability. The safety properties are *agreement* that states all non-faulty nodes output the same transaction and *total order* which indicates that all non-faulty nodes have identical sequence of transactions output. The liveness property states that if a transaction *tx* is input to *n-f* non-faulty nodes, then it is eventually output by every non-faulty node. The modular structure of the algorithm can be found in Figure 2.4.

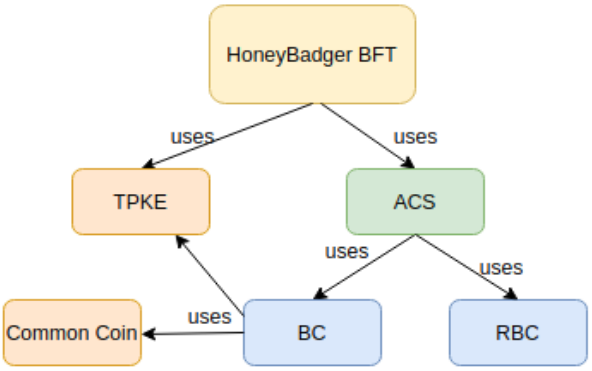

Figure 2.4: Module Relationships in Honey Badger BFT [18]

Asynchronous Common Subset (ACS) protocol is basically an asynchronous agreement on a common subset, which broadcasts the proposed value using reliable broadcast (RBC) and then based on what is returned by BC it decides a vector of the proposed value. Generally, *threshold public-key encryption* (TPKE) allows a set of users to decrypt a ciphertext if a given threshold of authorized users cooperates, with users here being the nodes for this

situation. Binary Consensus, Reliable Broadcast and Common Coin are basic protocols used by many BFT algorithms and are thoroughly explained in Chapter 3.

2.2.3. BEAT BFT Protocols

BEAT [19] presents a set of practical BFT protocols for completely asynchronous environments, as it consists of five (5) protocols, designed to meet different goals, like performance metrics or application scenarios, making it flexible and extensible. Also, BEAT leverages more secure and efficient cryptography support and more flexible and efficient erasure-coding support, as presented in Figure 2.5.

Taking a look at each algorithm, the baseline protocol BEAT0, incorporates more secure and efficient approaches, improving HoneyBadger. BEAT1 additionally replaces erasure-coded broadcast of HoneyBadger with Bracha's broadcast [2], with this reducing latency when there is low contention and a small batch size. After that, BEAT2 opportunistically moves the encryption part of the TPKE to the client, further reducing latency, however it does so at the price of achieving a weaker liveness notion. BEAT3 is a BFT storage system that replaces Byzantine reliable broadcast [2] with a more efficient primitive, and the improvement is significant, as it allows execution in bandwidth-restricted environments and significantly improves scalability. BEAT4 builds on top and further reduces read bandwidth by extending fingerprinted cross-checksums to handle partial read and the design of a novel erasure-coded asynchronous verifiable information dispersal protocol.

	Techniques	Features
BEAT0	more efficient labeled TPKE and coin flipping more flexible and efficient erasure coding support	baseline protocol outperform HB in all metrics
BEAT1	replacing AVID broadcast with Bracha's broadcast	latency optimized
BEAT2	client-side labeled CCA threshold encryption	latency optimized additional achieve causal order
BEAT3	a novel protocol combining AVID-FP and ABA	storage, bandwidth, throughput optimized
BEAT4	extending fpcc to single fragments and pyramid codes a novel AVID-FP saving bandwidth	further saving read bandwidth

Figure 2.5: Primary characteristics of BEAT protocols [19]

2.2.4. The Hashgraph Protocol

The Hashgraph protocol is an asynchronous BFT atomic broadcast protocol [20] that departs in structure from prior work by not using communication to vote, but only to broadcast transactions, by aiming to execute broadcast and voting simultaneously. It is based on a gossip protocol, in which the participants do not just gossip about transactions, they also gossip about gossip. That means, they jointly build a graph data structure called *hashgraph* reflecting all of the gossip events, as shown in Figure 2.6, maintaining the communication history among all nodes in the network and basically allowing Byzantine agreement to be achieved through virtual voting. Putting this into an example, server A does not send B a vote, but instead server B calculates what vote A would have sent, based on its knowledge of what A knows. This yields a fair Byzantine agreement, with less communication overhead beyond the transactions themselves.

An overview of how the algorithm works, starts with each node creating the graph locally and keeping the time each event was received. Then, it sorts the receival times and takes the median time, which reflects when the message reached half the processes (consensus time). Finally, these consensus times are being sorted and thus it has the same order of transactions among all nodes. However, in order for the construction of the graph to begin, each node that receives a transaction from a client, randomly chooses another server and sends the message and each process which receives the message does the same, so the communication explodes exponentially until everybody has each message.

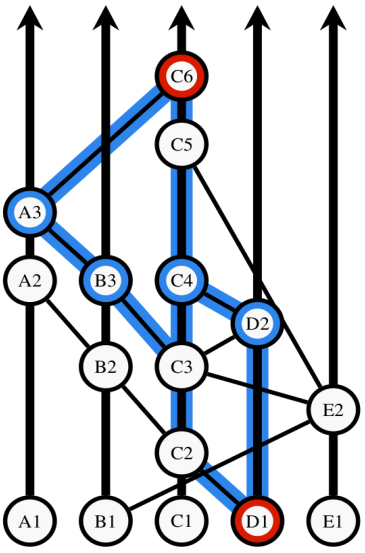

Figure 2.6: Common hashgraph data structure, with older events at the bottom [20]

2.3. The ZeroMQ Messaging Library

2.3.1. Introduction and Features

ZeroMQ (also known as ØMQ, 0MQ, or zmq) is a high-performance asynchronous messaging library [14], designed to be used in distributed and concurrent applications. ZeroMQ provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. The library's API is designed to resemble Berkeley sockets, and although it looks like an embeddable networking library, in reality it acts like a concurrency framework. Originally the zero in ZeroMQ was meant as "zero broker" or "zero latency", however since then, it has come to encompass different goals like zero cost or zero waste, and generally, "zero" refers to the culture of minimalism that permeates the ZeroMQ project.

Among its many benefits, ZeroMQ has sockets that carry atomic messages across various transport protocols like in-process, inter-process, TCP, or multicast, it has a score of language APIs and also runs on most operating systems. Furthermore, Zeromq is fast enough to be the fabric for any clustered product and its asynchronous I/O model can support scalable multicore applications, built for asynchronous message processing tasks. ZeroMQ was created and is being maintained and updated by iMatix Corporation [15] and is LGPLv3 open source.

2.3.2. Socket Types

One of the most important advantages of ZeroMQ is that it provides a range of sockets which generalize the traditional IP and Unix domain sockets, each of which can be combined and form N-to-N messaging patterns. Sockets provided by ZeroMQ are [16]:

- REQ: Sockets used by a client to send requests to and receive replies from a service. REQ sockets must follow the pattern send, receive, send, receive.
- REP: Sockets used by a service to receive requests from and send replies to a client. REP sockets must follow the pattern receive, send, receive, send.
- DEALER: Talks to a set of anonymous peers, sending and receiving messages using round-robin algorithms. Works as an asynchronous replacement for REQ, for clients that talk to REP or ROUTER servers.

- ROUTER: Talks to a set of peers, using explicit addressing so that each outgoing message is sent to a specific peer connection. Works as an asynchronous replacement for REP, and is often used for servers that talk to DEALER clients.
- PUB: Used by a publisher to distribute data. Messages sent are distributed to all connected peers. This socket type is not able to receive any messages.
- SUB: Used by a subscriber to subscribe to data distributed by a publisher. Initially a SUB socket is not subscribed to any messages. The send function is not implemented for this socket type.
- XPUB: Same as PUB except that you can receive subscriptions from the peers in the form of incoming messages.
- XSUB: Same as SUB except that you subscribe by sending subscription messages to the socket.
- PUSH: Talks to a set of anonymous PULL peers, sending messages using a round-robin algorithm. It has no receive operation.
- PULL: Talks to a set of anonymous PUSH peers, receiving messages using a fair-queuing algorithm.
- PAIR: Socket that can only be connected to a single peer at any one time. No message routing or filtering is performed on messages sent over a PAIR socket.
- CLIENT: Talks to one (1) or more SERVER peers. If connected to multiple peers, it scatters sent messages among these peers in a round-robin fashion, and it does not drop messages in normal cases.
- SERVER: Talks to zero (0) or more CLIENT peers. Each outgoing message is sent to a specific peer. A SERVER socket can only reply to an incoming message.

2.3.3. Messaging Patterns

Using and combining these socket types, various messaging patterns or architectures can be built depending on the topology needed, as in the example illustrated in Figure 2.7. ZeroMQ patterns are implemented by pairs of sockets with matching types. The core built-in messaging patterns [16] ZeroMQ offers are:

- Request-Reply: Connects a set of clients using a REQ or DEALER socket to a set of services using a REP or ROUTER socket. Messages sent to a service before the service becomes online are not lost, because they are stored in a queue, facilitating the preservation of messages.

- Publish-Subscribe: A remote distribution pattern that connects a set of subscribers using a SUB socket to a set of publishers using a PUB socket. Unlike the Request-Reply pattern, messages if not received are lost, and if the subscriber cannot keep up with the incoming messages then messages are dropped.
- Pipeline: Intended for task distribution, typically in a multi-stage pipeline where one (1) or a few nodes push work to many workers, and they in turn push results to one (1) or a few collectors. The pattern will not discard messages unless a node disconnects unexpectedly and it is scalable, as nodes can join at any time.
- Exclusive Pair: Intended for specific use cases where the two peers are architecturally stable. This limits its use within a single process for inter-thread communication, thus should be avoided to use in distributed applications.
- Client-Server: Used to allow a single server to talk to one or more clients. The client always starts the conversation, after which either peer can send messages asynchronously, to the other.

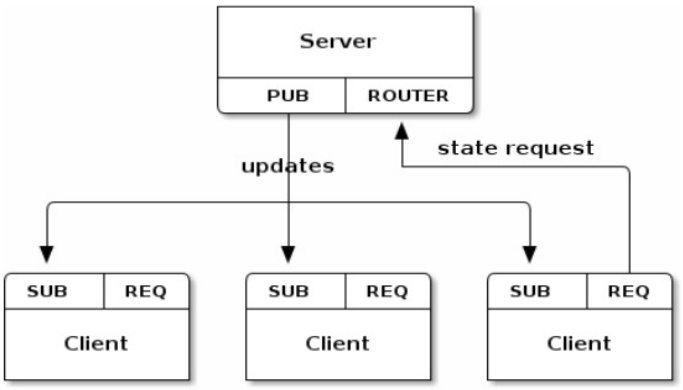

Figure 2.7: Example of a basic ZeroMQ usage [16]

2.4. The Go Programming Language

2.4.1. Introduction

Go (or Golang) is a statically typed, compiled programming language with incredibly fast compilation speeds designed at Google [13]. Some of its core developers were members of the UNIX team at Bell Labs like Ken Thompson and Rob Pike, and were primarily motivated by their shared dislike of C++. Primary goals were to have a programming

language that has static typing and run-time efficiency, that is readable and usable similar to Python and Javascript, trying to unify programming languages developers use within Google, and to have high-performance networking and multiprocessing capabilities.

2.4.2. Features

Go is designed to be syntactically similar to C, while also providing memory safety and garbage collection. Moreover, it provides some kind of Object Oriented Programming through the use of structural typing, structs and interfaces. Because it does not provide some of the main features of other OOP languages like inheritance and generics, it provides other features to make up for this. Some of them include type inference, a built-in remote package management system through a CLI program, and embedding which can be viewed as an automated form of composition or delegation. The best one comes through its interfaces, which provide runtime polymorphism. Interfaces are a class of types and provide a limited form of structural typing, basically an object which is of an interface type is also of another type, much like C++ objects being simultaneously of a base and derived class.

2.4.3. Concurrency in Go

The Go language has built-in facilities, as well as library support, for writing concurrent programs. Mainly, it deploys concurrency following the CSP paradigm, which is a formal language for describing patterns of interaction in concurrent systems. It does so by providing goroutines, channels, and a rich standard library package featuring most of the classical concurrency control structures.

Firstly, goroutines are light-weight coroutines, or more accurately green-threads, which are initiated with a function call prefixed with the “go” keyword, and so that function starts in a new concurrency “thread”. Channels provide the ability to send messages between goroutines, which are stored in a FIFO order that allows goroutines to wait either when they try to pull a message from an empty channel or when they try to push a message to a full channel. Therefore to avoid blocking on a full channel, the built-in “select” statement can be used to implement non-blocking communication on multiple channels. Thus, the Go programming language makes a perfect fit for parallel processing and networked systems.

Chapter 3

The Algorithm

3.1. Algorithm Structure

The problem can be stated this way: How does a set of distributed processes achieve consensus on a value, despite a number of process failures or Byzantine behavior? The algorithm's approach [1] in addressing this problem uses a stack of three Byzantine-resistant protocols aimed to be used in practical distributed systems, which are Multi-valued Consensus, Vector Consensus and Atomic Broadcast, as presented graphically in Figure 3.1. These protocols are designed as successive transformations from one to another. The first protocol, Multi-valued Consensus, is implemented on top of a randomized Binary Consensus and also uses a Reliable Broadcast Protocol. Moreover, the protocols assume that they are built on top of reliable channels, hence bit flips rarely happen.

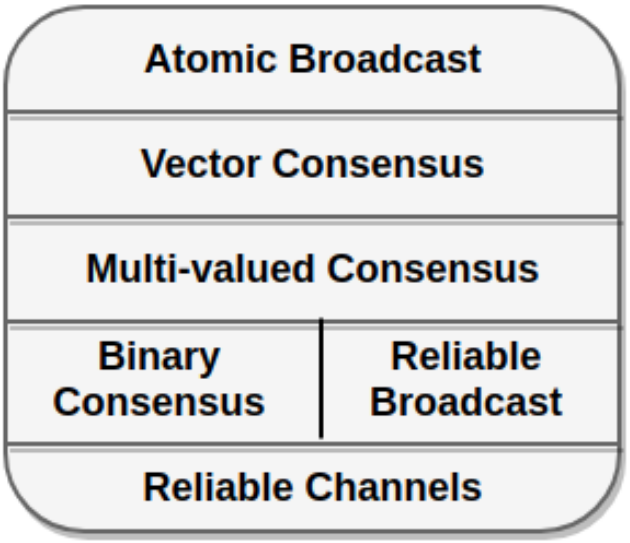

Figure 3.1: The algorithm's architecture [1]

The protocols share a set of supreme structural properties. Firstly and most importantly, as this is the main characteristic of the algorithm, the protocols do not use digital signatures constructed with PK cryptography, a well-known performance bottleneck in these kinds of protocols. Secondly, they are time-free, which means they make no synchrony assumptions, since these assumptions are often vulnerable to subtle but effective attacks and do not conform with real-world systems. Thirdly, they are completely decentralized, thus avoiding the cost of detecting corrupt leaders. Lastly, they have optimal resilience, as they tolerate up to a failure of $f=(n-1)/3$.

3.2. Protocols

We now give a more detailed description, starting from the lower layer and climbing to the top.

3.2.1. Reliable Channels

Reliable channels [4] are guaranteed by the ZeroMQ concurrency framework in our implementation, but they come with two structural properties; when the sender and the recipient of a message are both correct, then the message is eventually received and is not modified in the channel.

In practice, reliable channels have to be implemented using retransmissions to assure the first property and cryptography to make sure from whom a message is coming. For

cryptography, processes have to share symmetric keys, which it is assumed that are distributed before the protocol is executed through a trusted dealer. Therefore, the existence of the dealer does not collide with the protocol being decentralized, because the dealer has no role during the execution of the protocol.

3.2.2. Reliable Broadcast

The Reliable Broadcast (RBC) primitive [2] is used through the entire consensus procedure, by all nodes in order to disseminate their values and it is the basic building block of all consensus algorithms. The use of a reliable broadcast primitive ensures that a node can accurately send its value to a sufficient number of nodes, and also ensures essentially that all correct processes deliver the same messages, and that messages broadcast by correct processes are for sure delivered. Formally, a reliable broadcast protocol can be defined in terms of three (3) properties; validity, which states that if a correct process broadcasts a message *m*, then some correct process eventually delivers *m*, agreement, that says if a correct process delivers a message *m*, then all correct processes eventually deliver *m* and finally integrity that assures every correct process p delivers at most one message *m*.

The primitive is composed of four (4) basic steps, as seen in Figure 3.2, and its functionality is similar to the Three Phase Commit. Initially, each node sends an initial message to all others, informing them of its value. On step 1, nodes that received an initial message, echo that value to all others, which allows nodes to gather information regarding the number of nodes that have received the value in question. Upon receiving a total of $(n+f)/2$ echo messages for a value, each node broadcasts a ready message. Someone could state that step 2 is sufficient for the reliable broadcast to be terminated, however this is not the case, because while gathering the required number of echo messages, a node does not know that others have received enough echo messages too. To acquire that knowledge, the node would require to perform another step of communication, hence the step 3 in which a node waits for $2f+1$ ready messages to finally accept value v.

Step 0: Send (initial,v) to all other processes.

Step 1: Wait until the receipt of:
- one (initial,v) message,
- or (n+f)/2 (echo,v) messages,
- or (f+1) (ready,v) messages

for some v, and send (echo,v) to all other processes.

Step 2: Wait until the receipt of:
- (n+f)/2 (echo,v) messages,
- or f+1 (ready,v) messages

for some v, and send (ready,v) message to all other processes

Step 3: Wait until the receipt of 2f+1 (ready,v) messages for some v (including those received on steps 1 and 2), and **Accept v**

Figure 3.2: Reliable Broadcast Primitive [2]

3.2.3. Binary Consensus

A Binary Consensus (BC) protocol [2, 3] performs consensus on a binary value b $\in$ {0, 1}. The problem can be formally defined through three (3) properties; validity which expresses that if all correct processes propose the same value b, then any correct process that decides, decides b, agreement which states that no two correct processes decide differently, and finally termination that declares every correct process eventually decides. Instead of using a classic binary consensus algorithm [2], we used a more recent one [3] which seems to be much more efficient. The algorithm consists of a simple BVB abstraction as described in Figure 3.3 and the BC algorithm presented in Figure 3.4.

The broadcast is based on a particularly simple "echo" mechanism, but different from previous echo-based algorithms. When a process invokes $BVB(v)$, it broadcasts v to all the processes, and when a process p receives a message, if not yet done, it forwards this message to all the processes if it has received the same message from at least $f+1$ different processes. Finally, if p has received v from at least $2f+1$ different processes, the value v is added to a bin_values array, which can at the end contain either 0, 1 or both.

```
(00) broadcast v
(01) when (v) is received:
(02) if (v) received from (F+1) processes and not yet broadcast:
            then broadcast (v)
(03) if (v) received from (2F+1) different processes:
             then bin_values←bin_values ∪ {v}
```

Figure 3.3: Binary-value Broadcast algorithm [3]

The consensus algorithm can be decomposed in three (3) phases and runs in multiple rounds. This first phase is an exchange phase, in which a process p invokes BVB to inform the other processes of its initial estimate, associated with the round number and tagged as EST, and then *p* waits until bin_values is no longer empty. The second phase is also an exchange phase during which each correct process *p* broadcasts the value that belongs to bin_values tagged as AUX, to inform the other processes of estimated values that have been BV-broadcast by correct processes. Next, *p* blocks until the predicate of step 05 becomes satisfied, which is basically used to discard values sent only by Byzantine processes. The last phase is a local computation phase in which a correct process *p* firstly obtains the common coin value associated with the current round and if the values set contains both 0 and 1, *p* adopts the common coin value and proceeds to the next round, but if values set carries only 0 or 1, then *p* decides *v* if it is equal to the common coin otherwise, adopts *v* and moves to the next iteration.

```
est = v / r = 0

Phase 1
(01) r ← r + 1
(02) BVB (r, est, EST)
(03) wait until (bin_values = ∅)

Phase 2
(04) broadcast (r, w, AUX) where w ∈ bin_values
(05) wait until ∃ a set of (n-f) AUX messages delivered from distinct processes such that
     values ⊆ bin_values where values is the set of values x carried by these (n-f) messages

Phase 3
(06) s ←common_coin()
(07) if (values = {v})
(08)      then if (v = s) decide(v) else est← v
(10) else est← s
```

Figure 3.4: Randomized Binary Consensus algorithm [3]

3.2.4. Multi-valued Consensus

The first protocol of the stack proposed in [1] is a Multi-valued Consensus (MVC). The definition of the problem is similar to the binary consensus, except that processes can propose arbitrary values, not only binary. The protocol can decide one of the proposed values or a default value and can be expressed through its definition properties; validity1 that states if all correct processes propose the same value *v*, then any correct process that decides, decides *v*, validity2 which states that if a correct process decides *v*, then *v* was proposed by some process or $v=\perp$ and validity3 which declares that if a value *v* is proposed only by corrupt processes, then no correct process that decides, decides *v*. Agreement and termination properties are the same as for Binary Consensus.

Function MVC (v, cid):

Task T1
1. RB a (INIT,v,cid,i) message to all other processes
2. Wait until (n-f) INIT messages and store the proposed values in a vector
3. If (n-2f) messages with the same value v then
Select this value, otherwise the default value, as w
4. RB a (VECT,w,Vi,cid,i) message
5. Wait for (n-f) valid VECT messages and store the values received
6. If do not receive two VECT with different values and it also received
(n-2f) valid VECT msgs with the same value w:
Propose 1 for BA, else 0
7. If BC decides 0, then decide the default value
8. If BC decides 1, then wait for (n-2f) valid VECT msgs with the same value v and decide v.

Task T2
when m = (INIT, v, cid, j) is delivered
INIT_delivered ← INIT_delivered + m

Figure 3.5: Multi-valued Consensus algorithm [1]

Function MVC is always called with two arguments, the value proposed by the process and the consensus identifier as shown in Figure 3.5 above. During initialization, tasks T1 and T2 start concurrently, with T1 doing most of the work, while T2 simply receives *INIT* messages and stores them. Task T1 begins by reliably broadcasting an *INIT* message with the value *v* and then, the task waits for the reception of *n-f INIT* messages, including its own, and stores the proposed values in vector *V*. If a process receives at least *n-2f INIT*

messages with a value *v*, then it selects this value and reliably broadcasts with the vector V that justifies the selection, otherwise, it broadcasts the default value ⊥. After this, the process waits for *n-f* valid *VECT* messages, and if the process does not receive two *VECT* messages with different values , and it receives at least *n-2f* messages with *w*, it proposes 1 for the binary consensus, otherwise it proposes 0. If the binary consensus decides 0, the multi-valued consensus protocol decides on the default value ⊥. However, if the binary consensus decides 1, the process waits until it receives *n-2f* valid *VECT* messages with the same value *v* and decides *v*.

3.2.5. Vector Consensus

Next up is a Vector Consensus (VC) protocol which makes agreement on a vector with a subset of the values proposed, instead of a single value [1]. In systems where Byzantine faults can occur, the vector is useful only if a majority of its values were proposed by correct processes and therefore, the decided vector needs to have at least *2f+1* values. The protocol guarantees only that the majority of the values were proposed by correct processes, since in asynchronous systems messages can be arbitrarily delayed. Vector consensus can be defined in terms of the following properties: validity, which declares that every correct process that decides, decides on a vector *V* of size *n*, where *V[i]* is the value proposed by p_i or ⊥ and at least *f+1* elements of *V* were proposed by correct processes. Agreement and termination properties are also the same as in Binary Consensus.

The protocol is presented in Figure 3.6 and the arguments given are the value proposed and the identifier *(vcid)*. The protocol starts by reliably broadcasting the value proposed by the process alongside with the *vcid*. Afterwards, the protocol executes for one or more rounds until a decision is made, with maximum being *f* rounds because then all processes propose the same vector and all messages have already been delivered, because the algorithm begins each round by waiting for the reception of $n-f+round\_num$ messages cumulatively received since the beginning of the execution. Next, the process builds a vector *W* with the values it received from other processes and initiates Multi-valued Consensus with vector *W* and a unique combination of vcid and the round number as inputs. The protocol keeps executing until MVC returns a vector different from the default value ⊥, and then VC proposes that vector.

1. With RBC broadcast the value proposed.
2. r = 0
3. Repeat for one (1) or more rounds (max=f):
    a. Wait for (n-f+r) messages and store them in a vector.
    b. Propose this vector for the MVC
    c. When MVC return V ≠ ⊥, then decide V

Figure 3.6: Vector Consensus algorithm [1]

3.2.6. Atomic Broadcast

The problem of Atomic Broadcast (ABC), or Total Order Reliable Broadcast, is the problem of delivering the same messages in the same order to all processes and is implemented on top of the Vector Consensus protocol. The definition of the problem is equal to the definition of reliable broadcast plus a total order property therefore validity, agreement and integrity properties are exactly the same as for RBC. Total order states that if two correct processes deliver two messages, then both processes deliver the two messages in the same order.

Firstly, the initialization is carried out before the first transmission or reception of a message, and when ABC is called, it simply reliably broadcasts the message as Figure 3.7 indicates, and messages are handled by tasks *T1* and *T2*. When a message is received, it is inserted in the set *R_delivered*, so whenever this set is not empty, the process tries to agree with the other processes on the delivery of the messages in the set. The task starts by constructing a vector H with a hash of each message in *R_delivered*, which is essentially a fixed-length unique identifier of the message and the objective is to compress the input supplied to the Vector Consensus protocol. The VC decides on a vector *X* with at least *2f+1* vectors *H* from different processes. If the hash of a message appears in more than *f+1* of these vectors, the process can be confident that the hash was proposed by a correct process and not only by malicious processes. The process waits until all messages that are to be delivered are put in *R_delivered*, and then it stores them in *A_deliver*. Finally, the process delivers the messages in *A_deliver* in a pre-established order, removes them from *R_delivered*, and increments the atomic broadcast identifier by one.

Initialization
R_delivered←∅
aid←0
num←0
activate task (T1,T2)

When ABC (m) is called do
- RBC (num, m)
- num←num + 1

Task T1
(01) when (R_delivered not ∅) do:
(02) H←{hashes of the messages in R_delivered }
(03) X←VC (H, aid)
(04) wait until (all messages with hash in (f+1) or more cells in vector X are in R_delivered)
(05) A_deliver←{all messages with hash in (f+1) or more cells in vector X}
(06) Deliver messages in A_deliver in a deterministic order
(07) R_delivered←R_delivered - A_deliver
(08) aid←aid + 1

Task T2
when a message is delivered by RBC do
- R_delivered←R_delivered U {message}

Figure 3.7: Atomic Broadcast algorithm [1]

Chapter 4

Implementation

4.1. Implementation Decisions

As previously mentioned, for the implementation of the asynchronous randomized BFT algorithm we used Go as the programming language, embedded with the ZeroMQ messaging library. The implementation is open for everyone on GitHub [24, 25].

The reasons why we chose Go as our programming language are many. Firstly, there is a lot of hype around Go for being the de facto language for concurrent and parallel systems [13], as Go is packed with great and strong parallelism tools like goroutines and channels. Since we needed to have each processor running concurrently, goroutines greatly facilitated the need for multi-threading support, and channels are the means to connect and synchronize concurrent goroutines. Furthermore, our results will be compared with the results of similar algorithms which were developed also using Go, thus having them all implemented in the same programming language makes the comparison and evaluation of the algorithms much more valid and accurate. Finally, even though Go is a newly developed programming language, due to the fact that it is delivered by Google and is really hyped in the market, its documentation on the Internet is really huge and remarkably helped us during the implementation process.

Additionally, we decided to go with the ZeroMQ as our messaging library because it is a light-speed asynchronous messaging library [14] that provides a variety of sockets that can be used to deploy a vast amount of distributed messaging patterns in any networked topology required. ZeroMQ is state-of-the-art in terms of speed, and reliability and by using hidden message queues makes the delivery of messages guaranteed. Moreover, ZeroMQ follows a simple interface that greatly facilitates usability of the library and the fact that it is open source makes it the ideal messaging framework. Finally, the compared implementations also used ZeroMQ, therefore the comparison would only benefit if we used the same libraries as well.

4.2. Project Structure

The implementation consists of two (2) different Go projects, with the first one imitating the processor's behavior and the other one the client's. The structures are illustrated in Figures 4.2 and 4.7 respectively. Since Go does not allow cyclic dependencies between packages, the files for the algorithm's modules are located in the same package, called *modules* for server and *app* for client, as these modules interact with each other by calling each other's interface functions. In the *config* folder, the code for configuring the IP addresses of the system as well as for configuring the execution scenario are located. Additionally, a *logger* is implemented that writes info and errors in text files for the monitoring of the system's operation. The *types* package contains all the necessary Go *structs* and type aliases required for the messages that are exchanged among the servers and the clients. Also, the *variables* package contains the main variables and constants that are shared between modules, such as the number of processors, Byzantine nodes and clients. Finally, *messenger* is the package responsible for sending and receiving messages between clients and servers and among the processes as well.

4.3. Communication

For the implementation of the messenger we used the messaging framework ZeroMQ as already mentioned, and also we based the implementation on another similar thesis [23]. In

detail, each processor has a REQ/REP pair of sockets to communicate (request/reply) with other processors and a REP socket for each client, in which it receives requests. Furthermore, each server node adds an extra PUB socket connected with each client to send the reply back to them. Clients deploy a REQ along with a SUB socket for each processor, as shown in Figure 4.1, in order to transmit their request through the REQ socket and receive responses after a while through SUB.

This way processors intercommunicate with the use of their corresponding pair, while clients send requests to processors through a REQ/REP pair, only now the processor will immediately reply with an empty message, rendering its REP socket available for the next request, and replies are being sent through the PUB socket. With this solution, there will not be cases where REP sockets try to receive twice consecutively, or REQ sockets trying to send twice, an error which would otherwise cause an error and crash the system.

Thus, with this solution we have avoided a case where a client might block waiting for a server's response or vice-versa. Also, even though REQ/REP sockets are synchronous we used them due to the fact that they are the most reliable ones, and to add asynchrony we used Go to have a form of a timeout and retransmission of messages. This was achieved by using Go channels and the "select" statement alongside the timeticker timeout functionality that Go offers.

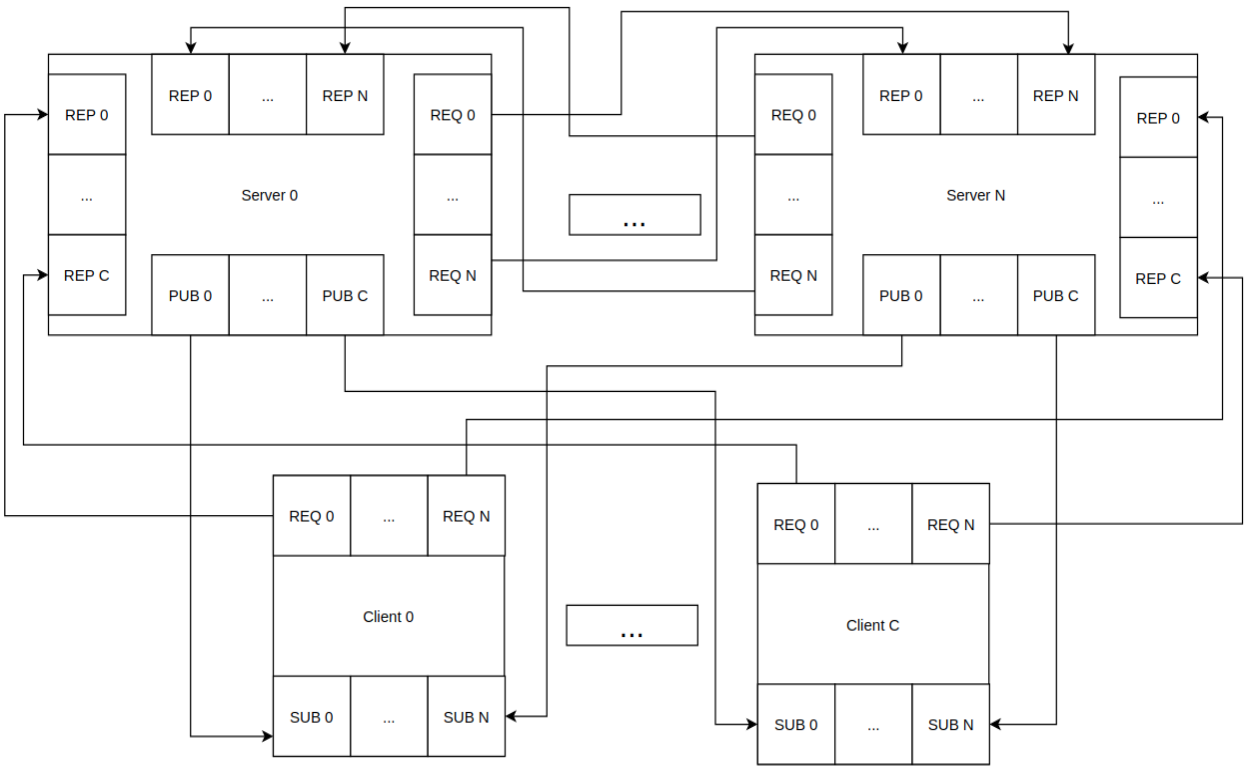

Figure 4.1: Communication structure with ZeroMQ sockets

4.4. Server Implementation

After the above overview on the project's structure, let us have a closer look specifically at the server implementation, presented in Figure 4.1. Processors send and receive messages among other processors and their clients through concurrent goroutines and basically execute the algorithm explained in Chapter 3 in order to achieve Atomic Broadcast for the clients requests. Each process holds a character array in which the client requests are placed, and the goal is to have consistent arrays in all nodes, which will be accomplished with the BFT algorithm [1], therefore we need consensus on these character arrays.

When a server is launched, the sockets are being initialized as well as the goroutine functions that send and receive the messages. Moreover, the variables are being initialized with the values given from the command line on launch, like N, server's ID, number of clients, scenario number and whether the execution is local or in real-world. After, the Atomic Broadcast module is initiated and finally the Request Handler that is responsible for handling the clients requests by providing them to ABC. The server runs indefinitely or until it is shut down.

- BFTWithoutSignatures
  - modules
    - request_handler.go
    - atomic_broadcast.go
    - vector_consensus.go
    - multivalued_consensus.go
    - binary_consensus.go
    - reliable_broadcast.go
  - messenger
    - messenger.go
  - config
    - ip.go
    - local.go
    - scenario.go
  - types
    - message.go
    - abc_message.go
    - vc_message.go
    - mvc_message.go
    - bc_message.go
    - rb_message.go
    - client_message.go
    - reply_message.go
  - variables
    - variables.go
  - threshenc
    - key_generator.go
    - key_reader.go
    - sign_and_verify.go
  - logger
    - logger.go
  - main.go

Figure 4.2: Server program structure

4.4.1. Modules

In the *modules* folder as already mentioned, we have implemented the algorithms presented in Chapter 3 one by one. For the development we started from the bottom of the stack of protocols and went to the top, thus making it easy to build on top of the previously implemented algorithm. Firstly, we have the Binary Consensus [3], which consists basically of two (2) algorithms, BC and BVB, with BC being the main one that executes the consensus procedure (Figure 3.4) and BVB being the algorithm that broadcasts the messages and fills the bin_values set (Figure 3.3). For the randomized BC algorithm we needed a Common-Coin algorithm [22], however we simply use BC round to acquire it (*bcid%2*), which might not be Byzantine tolerant but it does not affect our implementation at all as we are not going to have Byzantine nodes attacking the common coin itself at this point. However, several common coin algorithms exist in the bibliography. Moreover, in many occasions in the algorithms, we needed to uniquely combine two (2) numbers, like for Binary Consensus the bc identifier and the round number, therefore we used Cantor's pairing function [26], which is basically this equation:

```
((a * a) + (3 * a) + (2 * a * b) + b + (b * b)) / 2
```

Next, we implemented the Reliable Broadcast algorithm (Figure 3.2) which is used by all other protocols above in the stack in order to reliably broadcast the messages between the nodes. Afterwards, we developed the code for the three (3) main protocols of the algorithm, MVC, VC and ABC just like they are presented in Figures 3.5, 3.6 and 3.7 respectively. Finally, we implemented the Request Handler (Figure 4.3) which is responsible for receiving the requests from the clients and calling the ABC, with message m being the client's request. When the ABC delivers some values, it adds them to the array, the main consensus object, and then replies to the corresponding client that its request has been delivered.

```
var (
	Aid = 0

	Array = make([]rune, 0)   // Array - The array that has to be in consensus
)

func RequestHandler() {
	// Accepts the requests from the clients and calls ABC
	go func() {
		for message := range messenger.RequestChannel {
			AtomicBroadcast(message)
		}
	}()

	// Gets result from ABC, appends it in the Array and replies to the client
	go func() {
		for message := range Delivered {
			for _, v := range message.Value {
				var m types.ClientMessage
				buffer := bytes.NewBuffer(v)
				decoder := gob.NewDecoder(buffer)
				err := decoder.Decode(&m)
				if err != nil {
					logger.ErrLogger.Fatal(err)
				}

				Array = append(Array, m.Value)

				go ReplyClient(types.NewReplyMessage(m.Num), m.Cid)
			}
		}
	}()
}
```

Figure 4.3: Request Handler sample code

4.4.2. Messenger

In the *messenger* folder, the initialization of the sockets takes place as well as some basic functions that broadcast the messages to other processors, receive messages and requests and reply to clients. Except for these, we also have a couple of functions that in case we

are in a scenario that not all servers act non-faulty, the messages sent by Byzantine nodes are modified before transmission to try and harm consensus among the processes.

4.4.3. Config

For local development, the mapping of network addresses to nodes happens in the *local.go* file. It contains four maps of integers to strings, each map storing the corresponding address of the appropriate socket type, and we play with the localhost and different ports representing each processor or client, as illustrated in Figure 4.4. Similarly, for the real-world configuration, instead of tinkering with ports, each processor has its own computer and IP address found in the *ip.go* file.

Finally, *scenario.go* is responsible for configuring the scenario of execution the servers run, which is basically a flag that allows us to use it in our code and determine how a process needs to act.

```
func InitializeLocal() {
        RepAddresses = make(map[int]string, variables.N)
        ReqAddresses = make(map[int]string, variables.N)
        ServerAddresses = make(map[int]string, variables.Clients)
        ResponseAddresses = make(map[int]string, variables.Clients)

        for i := 0; i < variables.N; i++ {
            RepAddresses[i] = "tcp://*:"+strconv.Itoa(4000+(variables.ID*100)+i)
            ReqAddresses[i] = "tcp://localhost:"+strconv.Itoa(4000+(i*100)+variables.ID)
        }

        for i := 0; i < variables.Clients; i++ {
            ServerAddresses[i] = "tcp://*:"+strconv.Itoa(7000+(variables.ID*100)+i)
            ResponseAddresses[i] = "tcp://*:"+strconv.Itoa(10000+(variables.ID*100)+i)
        }
}
```

Figure 4.4: IP configuration for local execution (local.go)

4.4.4. Types and Variables

Since most of the messages that are exchanged in the system are complex structures containing multiple fields, and ZeroMQ sends messages as a sequence of bytes, a method of serialization and deserialization of structs is necessary, so for this requirement we used

Gob [27]. Gob is a package included in Go's standard library, and it manages streams of bytes exchanged between a transmitter and a receiver. Encoding with Gob returns an array of bytes while decoding with Gob builds a struct from an array of bytes. Gob supports encoding and decoding of all Go's built-in types, but to be able to encode/decode a complex struct, that struct has to implement the GobEncoder/GobDecoder interface by implementing the two (2) basic methods.

So here comes the *types* folder, in which we implemented all the basic types of messages we need for our implementation, like abc_message or client_message (Figure 4.5). The message.go is the general type in which processors' messages are wrapped. Its structure is composed of the fields Payload, which is an array of bytes, a string Type that denotes the type of the payload in terms of the name of the structure, and an integer From denoting the identity of the sender processor. When a server receives an incoming message, it decodes received bytes into a Message struct, and then a switch case is applied on the structure's Type field. The message's payload is decoded as well to the appropriate struct type, and the resulting struct is consequently written to the corresponding channel, in which the consumer module will read from. This pattern succeeds in hiding delays of sending and receiving messages, making the message exchange feel more organic.

Additionally, the *variables* folder just consists of a file that contains all the vital variables and constants that are used generally in the project. Some of them are, the processor's ID, the number of servers, clients and Byzantine nodes in the system, the MVC and VC default value and whether the execution is local or in real-world.

```
// AbcMessage - Atomic Broadcast message struct
type AbcMessage struct {
	Num   int
	Value []byte
}

// GobEncode - Atomic Broadcast message encoder
func (abcm AbcMessage) GobEncode() ([]byte, error) {
	w := new(bytes.Buffer)
	encoder := gob.NewEncoder(w)
	err := encoder.Encode(abcm.Num)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	err = encoder.Encode(abcm.Value)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return w.Bytes(), nil
}

// GobDecode - Atomic Broadcast message decoder
func (abcm *AbcMessage) GobDecode(buf []byte) error {
	r := bytes.NewBuffer(buf)
	decoder := gob.NewDecoder(r)
	err := decoder.Decode(&abcm.Num)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	err = decoder.Decode(&abcm.Value)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return nil
}
```

Figure 4.5: Example of GobEncode and GobDecode for ABC type of message

4.4.5. Threshenc

Moving now to the *threshenc* folder, it is what is called Trusted Dealer. Basically, in this folder using built-in libraries of Go, Verification and Secret key-pairs (or Public and Private), for each one of the servers are being created before the processors start their

execution, and these key-pairs are stored locally. Then, when each node starts its execution it reads and keeps the Verification keys of all the servers but only its own Secret key, that is why it is called secret. Therefore, the overhead that the keys have on each processor is just the time it takes to read them from the local files and not their generation too, as that procedure takes place before the execution.

The reason why these key pairs are needed even though the implemented algorithm does not use any digital signatures or threshold encryption schemes, is simply to have a method of validation for each message that is received, to guarantee that the sender is valid. Therefore, using the *sign_and_verify.go* file (Figure 4.6), each node signs the message before sending using their Secret key, appends the signature to the message and then sends it. When the message is delivered, the receiver process verifies that the signature is valid using the Verification key of the sender. If the verification procedure of the message is correct, then the message is consumed from the algorithm, but if it is not correct then it means it is a malicious one and thus it is dropped immediately.

```
func SignMessage(message []byte) []byte {
	hash := sha256.New()
	_, err := hash.Write(message)
	hashSum := hash.Sum(nil)

	signature, err := rsa.SignPSS(rand.Reader, SecretKey, crypto.SHA256, hashSum, nil)
	return signature
}

func VerifyMessage(message []byte, signature []byte, i int) bool {
	hash := sha256.New()
	_, err := hash.Write(message)

	err = rsa.VerifyPSS(VerificationKeys[i], crypto.SHA256, hash.Sum(nil), signature, nil)
	if err != nil {
		return false
	}
	return true
}
```

Figure 4.6: Sign and Verify methods

4.5. Client Implementation

Similarly with the Server Implementation above, the client's program has a project structure consisting of *messenger, config*, *types*, *variables* and *logger* folders that actually serve the exact same purpose like in the server's project, but with fewer complexity as client's code is much simpler and does not implement any specific algorithm.

The client sends a request that contains a character to *f+1* servers, in order to be sure that at least one correct got the request, and then awaits for *f+1* ACK responses from different nodes before accepting that the request was successfully delivered and moving on sending the next request, for the same exact reason.

When a client is launched, the sockets are being initialized as well as the goroutine functions that send requests and receive replies. Moreover, the variables are being initialized with the values given from the command line on launch, like *n*, *f*, client's ID and whether the execution is local or in real-world. After, the client's main program is initiated that is responsible for setting and transmitting the requests and receiving the ACK messages. The client also runs indefinitely or until it is shut down.

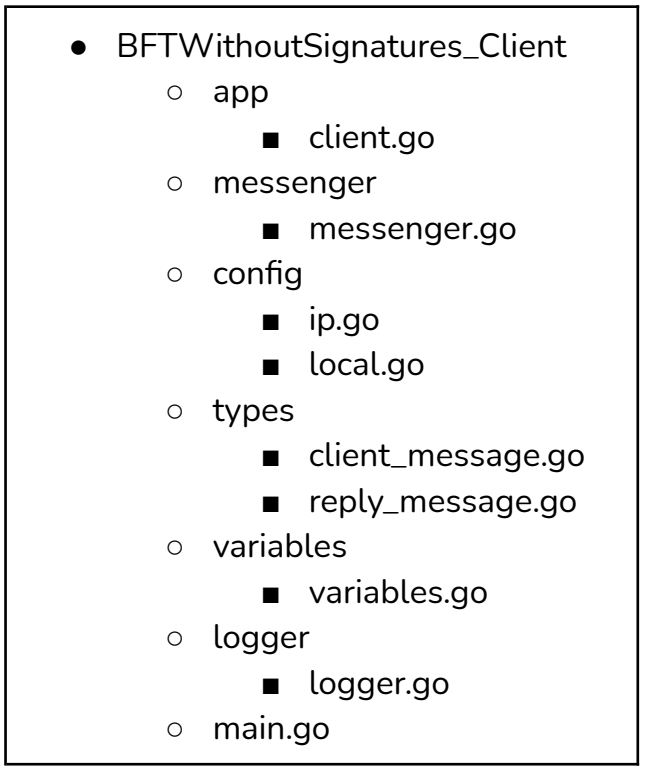

Figure 4.7: Client program structure

In the client.go file the actual logic of the client's implementation is located. Firstly, the client blocks for some seconds before sending its initial request to avoid having all clients sending a request at the same exact time initially. With the help of a pseudo-random generator that Go offers, the client selects "randomly" a readable character and *f+1* servers to transmit its request to. When the client submits its request, it waits for *f+1* ACK replies from different server nodes to verify that its request was truly delivered and then sends a new request following the exact same procedure over and over again. Additionally, each client simultaneously calculates the average operation latency by keeping track of the time it took for *f+1* servers to respond to their requests.

```
func Client() {
   rand.Seed(int64((variables.ID + 3) * 9000))                // Pseudo-Random Generator
   time.Sleep(time.Duration(variables.ID) * time.Second) // Wait a bit before sending 1st request
   sendRune()

   go func() {
      for message := range messenger.ResponseChannel {
         if _, in := replies[message.Value][message.From]; in {
            continue    // Only one value can be received from each server
         }
         if replies[message.Value] == nil {  replies[message.Value] = make(map[int]bool) }
         replies[message.Value][message.From] = true

         // If more than F+1 with the same value, accept the array
         if len(replies[message.Value]) >= (variables.F+1) && !accepted[message.Value] {
            accepted[message.Value] = true
            OpLatency += time.Since(sentTime[message.Value])
            sendRune()
         }
      }
   }()
}

func sendRune() {
    num++
    message := types.NewClientMessage(variables.ID, num, runes[rand.Intn(len(runes))])

    for i := 0; i < (variables.F + 1); i++ {
        to := (rand.Intn(variables.N) + i) % variables.N
        messenger.SendRequest(message, to)
    }
     sentTime[num] = time.Now()
}
```

Figure 4.8: Client main operation sample code

Chapter 5

Experimental Assessment

5.1. Experimental Environment

For our experimental evaluation we executed our system distributedly. Basically, we used several machines that worked as our server nodes and another machine that represented the clients. Having them running in parallel and while connected through their public IP via ZeroMQ socket API, the clients and servers worked as a distributed system that was Byzantine Fault Tolerant due to the algorithm they served. The machines we used were a cluster of nine (9) hosts and their basic CPU resource characteristics are presented in Figure 5.1. Based on the scenario we ran and the number of servers and clients we needed, each machine was running one (1) server instance, except from our two (2) best machines, *0* and *1*, that were executing all the extra instances, when servers were more than eight (8) in our experiments as well as for the seven (7) servers execution that we only used machines *0, 1, 6,* and *7*. The client processes ran on a different machine alone, and only when the clients were 50 and 75, we split them up to two (2) and three (3) machines respectively to have a maximum of 25 clients running on each machine.

Machine ID	CPU(s)	Threads per core	Core(s) per socket	Socket(s)	CPU Model
0	8	1	4	2	Intel(R) Xeon(R) CPU
1	8	1	4	2	Intel(R) Xeon(R) CPU
2	1	1	1	1	AMD Opteron(tm) Processor 252
3	2	1	1	2	AMD Opteron(tm) Processor 252
4	1	1	1	1	AMD Opteron(tm) Processor 252
5	1	1	1	1	AMD Opteron(tm) Processor 252
6	2	1	1	2	AMD Opteron(tm) Processor 246
7	2	1	1	2	AMD Opteron(tm) Processor 252
8	1	1	1	1	AMD Opteron(tm) Processor 252

Figure 5.1: Machines' basic CPU characteristics

5.2. Experiment Scenarios

The experiments we ran were a series of scenarios, with some of them having Byzantine nodes acting arbitrarily (presented in the next sections), which were constructed with the intention of evaluating the following three (3) measurements:

- The **average message complexity** of the processors through a round of Atomic Broadcast. In other words, the average number of messages each node sends in a round, before delivering a result.
- The **size of the messages** that were sent in a round of Atomic Broadcast, calculated in MB. Basically, the total size of all the messages sent in one ABC round.
- The **average time** servers took to deliver a client request in the consensus array. Also referred to as **operation latency**, is the average of the time duration from the moment a client sends its request until it receives $f+1$ ACK messages.

5.2.1. Failure-free Operation Scenario

It is natural that our base scenario must be the one where no Byzantine behavior exists. Fundamentally, in this scenario we have the Byzantine nodes acting as correct, so that we take the "best" available above mentioned metrics and then be able to compare them with

the next scenarios where Byzantine behavior exists in several forms. Also, this base scenario is common among many of the similar BFT algorithms in the bibliography, as mentioned in Chapter 2, thus making it easy for us to compare our results with theirs.

5.2.2. Byzantines Idle Scenario

The second scenario is another vital one in which we have the Byzantine processors not sending or broadcasting anything, expressly remaining completely idle. To put it another way, the Byzantine processes listen for any incoming requests from clients or messages from other servers, however they do not participate at all in the consensus procedure, by not delivering any results, replying to the clients and broadcasting messages. This is an excellent scenario because it stresses the algorithm and confirms that the system keeps consensus as the algorithm tolerates this behavior from f processes.

5.2.3. Binary Consensus Attack Scenario

In the next scenario, we have the Byzantine nodes trying to attack the Binary Consensus algorithm specifically. When a Byzantine server needs to send a BC message, it modifies it before the transmission in an effort to confuse correct servers and achieve a delay in the consensus process due to the fact that correct nodes might need to send more messages to agree on the binary value, or even break consensus something that it is not achieved as the algorithm tolerates this behavior from up to f nodes. The way Byzantines modify the messages is based on the BC round and the receiver of the message, they send a wrong binary value to all nodes or a correct to half the processes and a wrong to the others.

5.2.4. Half and Half Attack Scenario

Last scenario is what we called the Half and Half Attack, which has a similar approach as the previous one. Roughly speaking, the Byzantine nodes modify all the messages before sending them and send one value to half and another to the others. To be exact, when a faulty processor needs to send any type of message, from BC, to ABC, to replying to the client, the server adjusts the message so to transmit a message to everyone, but a different one to the servers with odd ID and the ones with even ID. Opting for simplicity, faulty messages are just one byte, 0 or 1 characters, correct messages that are dozens of bytes, but

nevertheless they still accomplish generating confusion among the correct nodes without hurting consensus though.

5.3. Experiment Results and Analysis

5.3.1. Theoretical Performance Evaluation

Firstly, we will illustrate the theoretical perspective of the performance of the algorithm as this is given in the original paper [1], and presented in Figures 5.2 and 5.3. The time complexity of the Multi-valued Consensus protocol is twice the number of asynchronous rounds executed by the Reliable Broadcast protocol plus the time complexity of the Binary Consensus. The RBC protocol [2] runs in exactly three rounds, while the time complexity of the BC protocol [3] is measured in the expected number of asynchronous rounds, since the protocol is randomized and probabilistic, however it has constant expected time complexity *O(1)*. Therefore, the time complexity of the MVC protocol is overall *O(1)*. Moreover, the Vector Consensus protocol runs in the best case in one round, and in the worst in *f+1* rounds, so in the pessimistic assumption the expected time complexity of the algorithm *O(f)*. Finally, the time complexity of the Atomic Broadcast protocol is equivalent to one (1) RBC plus one (1) VC and thus the expected number of rounds is *O(f)* per message.

Protocol	**Best time complexity**	**Expected time complexity**
Multi-valued Consensus	lmvc = 2lrb + lbc = 16	Lmvc = 2Lrb + Lbc = 26 = O(1)
Vector Consensus	lvc = lrb + lmvc = 19	Lvc = Lrb + (f+1)Lmvc = O(f)
Atomic Broadcast	lab = rb + lvc = 22	Lab = Lrb + Lvc = O(f)

Figure 5.2: Time complexities of the three protocols in asynchronous rounds [1]

As mentioned in [1], message complexities differ if the communication is point-to-point or broadcast. Point-to-point links are a connection between individual pairs of machines, while Broadcast links are a communication channel that is shared by all the machines in the network. So, the difference between point-to-point and broadcast, is that in broadcast networks, the packets are sent by any machine and received by all the other machines. If

the communication is point-to-point, the message complexity of Reliable Broadcast [2] is $2n^2+n$ and the expected message complexity of the Binary Consensus [3] is $O(n^3)$. On the other hand, for broadcast the complexities are $2n+1$ and $O(n^2)$ respectively. Therefore, the message complexity of Multi-valued Consensus corresponds to $2n$ executions of RBC plus one (1) BC that equals $O(n^3)$ for point-to-point and $O(n^2)$ for broadcast. The Vector Consensus has consequently $O(fn^3)$ for point-to-point and $O(fn^2)$, with Atomic Broadcast having the same message complexities as well.

Protocol	**Expected message complexity (point-to-point)**	**Expected message complexity (broadcast)**
MVC	Mmvc = 2nMrb + Mbc = $O(n^3)$	M'mvc = 2nM'rb + M'bc = $O(n^2)$
VC	Mvc = nMrb + (f+1)Mmvc = $O(fn^3)$	M'vc = nM'rb + (f+1)M'mvc = $O(fn^2)$
ABC	Mab = Mrb + Mvc = $O(fn^3)$	M'ab = M'rb + M'vc = $O(fn^2)$

Figure 5.3: Message complexities of the three protocols [1]

5.3.2. Experimental Results

We executed the client and server processes as they were presented in Chapter 4, specifically Sections 4.4 and 4.5, in the environment we mentioned above in order to gather the three (3) metrics; operation latency, message complexity and size. Unfortunately, our results were not the best possible due to the hardware resource bottlenecks we faced, but even with these, the outcome is likely promising. Basically, we focused on two main experiment situations to test our program; on the first one we kept the same number of servers, four (4), and kept increasing the number of clients sending requests, while on the other one we always had 25 clients and kept increasing the number of servers (n). All the results are presented below in Figures 5.4 to 5.9.

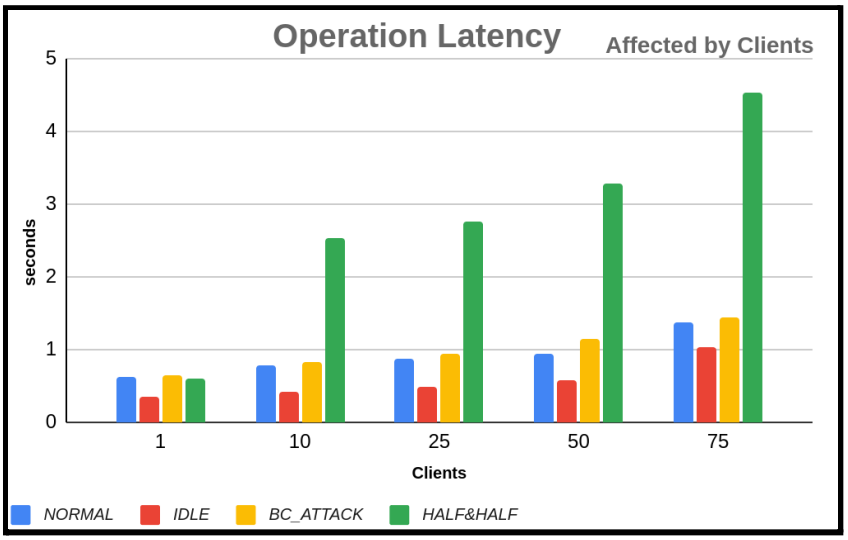

Figure 5.4: Operation Latency affected by the increase of clients

Starting with the first situation we are going to exploit, the metrics being affected by the increment of the clients in the system, and especially with the operation latency, we can surely identify a pattern. We can see that when the client is just one (1) the latency is similar for all four (4) scenarios as we can easily understand that the requests from a single client are not enough for giving the chance to Byzantine nodes to force correct ones to slow down. As the clients begin to increase in our experiments, we notice that generally the operation latency is also being increased as a consequence of the more requests that processors need to serve. However, we see that in the second scenario where Byzantines remain idle, the operation latency is always almost half the normal one due to the much less messages being exchanged among processes as faulty servers do not broadcast anything and thus the congestion in the network is less. Finally, we observe that the binary consensus attack from Byzantines does not affect the operation latency at any level, but in the Half and Half attack scenario the latency is really being affected, causing a big delay to the servers' response. In general, the normal scenario's latency is not being altered much with the rise of the clients, but remains pretty low at just over one (1) second.

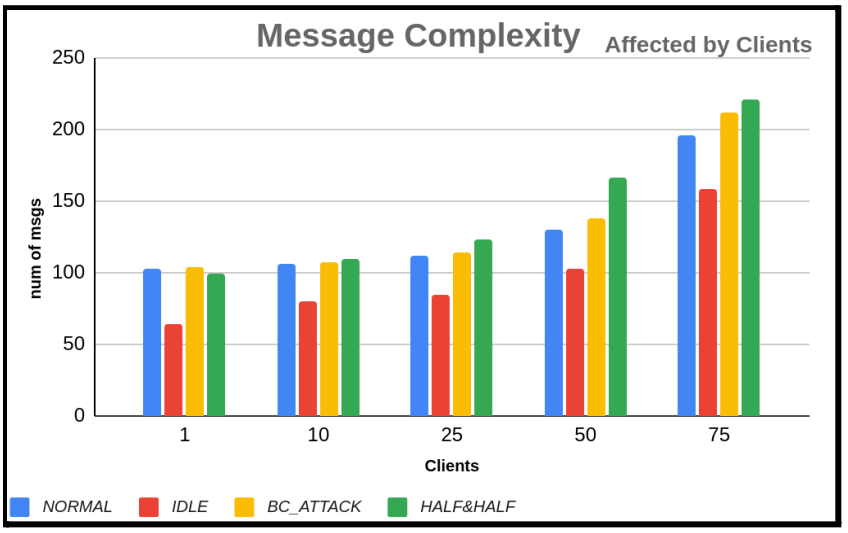

Figure 5.5: Message Complexity affected by the increase of clients

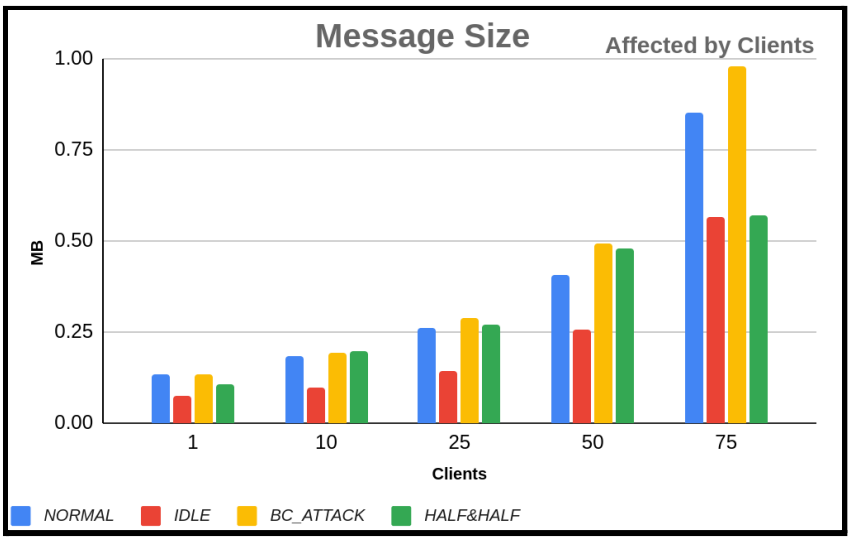

Figure 5.6: Message Size affected by the increase of clients

Let us now continue with the message complexity and size metrics. Here, we can also see that for the idle Byzantines scenario, both complexity and size are less than in the normal one because the faulty nodes do not broadcast any values at all, and so the averages are lower. Again, like for the operation latency, normal and BC attack scenarios have really similar values for both size and complexity through the increase of the clients. For the fourth scenario, we see that even though clients rise and the message complexity increases, the size is less than the normal scenario. That is due to the fact that half of the messages sent by Byzantine nodes are just one byte but the real messages are some dozens of bytes and that really decreases the average size. Overall, we can see that both message

complexity and size are being increased steadily as the clients grow and range between 50 - 250 messages and 0.15 - 0.9 MB respectively.

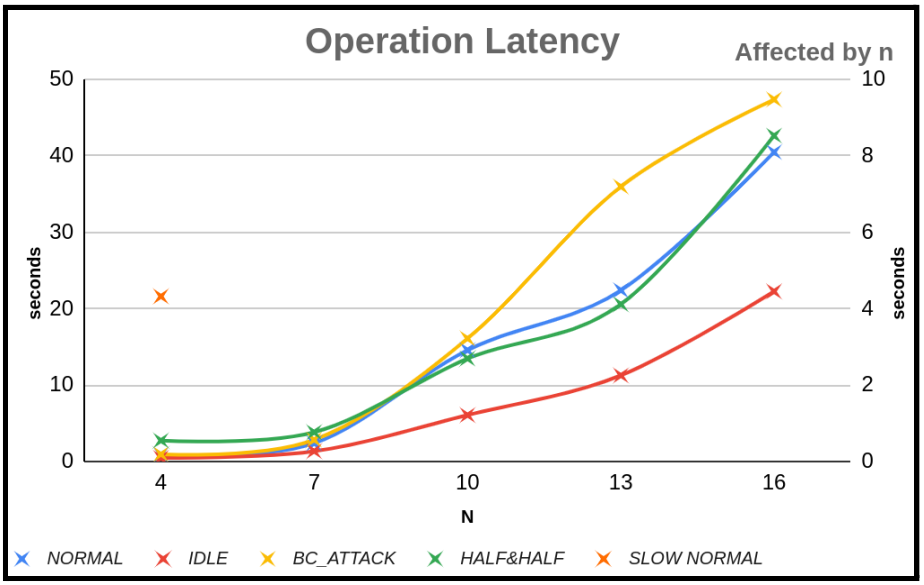

Figure 5.7: Operation Latency affected by the increment of n

On the other hand, in the situation where the number of clients remains the same and processes in the system increase, the results are a little different. On the primary axis we have the latencies of the four (4) scenarios, while on the secondary axis we only have the latency of the failure-free scenario running with the four (4) worst machines in the cluster where we ran the experiments (orange mark). Initially, we can notice that in the Idle scenario, servers always respond to the clients at half the time they do to respond in the other scenarios, and that is due to the fact that the message congestion in the network is much less here scenario as explained better above. The major difference from the previous situation comes in the BC attack scenario which now as the nodes increase it seems to affect the execution. Basically, we can see that the average latency for 13 and 16 nodes for the third scenario is a bit larger, which indicates that if we keep increasing the processors this difference will still be there or grow even more.

In contrast, we now see that the Half and Half attack scenario does not really have an effect on the latency of the system and has almost identical latency as the failure-free scenario. Generally, we can observe that the Failure-Free scenario has a kind of an exponential growth which is not the best case. The reason for this is primarily the lack of good resources to run the servers, which causes the latency to increase this much by just adding one more Byzantine server to the system. We can see that by also comparing it to

the theoretical latency described in the previous section. While for four (4) and seven (7) servers the results seem to be exactly what we want, around one (1) and two (2) seconds respectively, for larger clusters the delay is huge and that is just mainly resource related, because we had to also employ some slower and low on resource machines to execute these experiments, that of course had a negative impact on our results due to their bottleneck.

As mentioned in Section 5.1., for four (4) and seven (7) processes we used the best possible machines of the cluster to obtain these results, however, for 10, 13, and 16 we used all the available machines. Therefore, if we compare the results for the failure-free scenario with four (4) servers, the one with the best ones (blue) and the other with the worst ones (orange), we can see that operation latency on the worst machines is four (4) times slower than the one on the best machines. Consequently, we can realise that the processing of the messages on the worst machines is much slower than on the best. So we can see that when we add these machines on the other experiments, with the 10, 13 and 16 nodes, we can fully understand why the latency exponentially increased instead of following the theoretical evaluation. Evidently, the low processing power of the machines is causing the operation latency to grow by a lot.

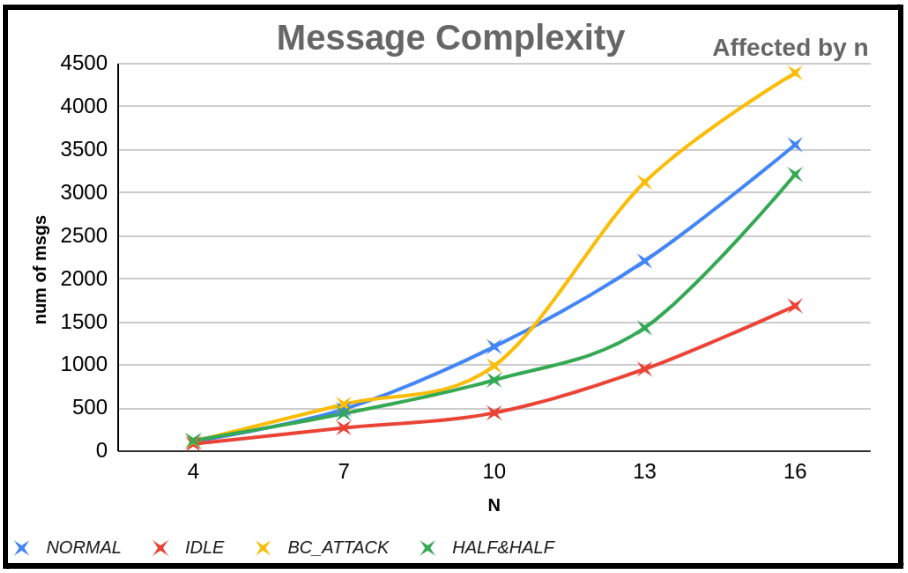

Figure 5.8: Message Complexity affected by the increment of n

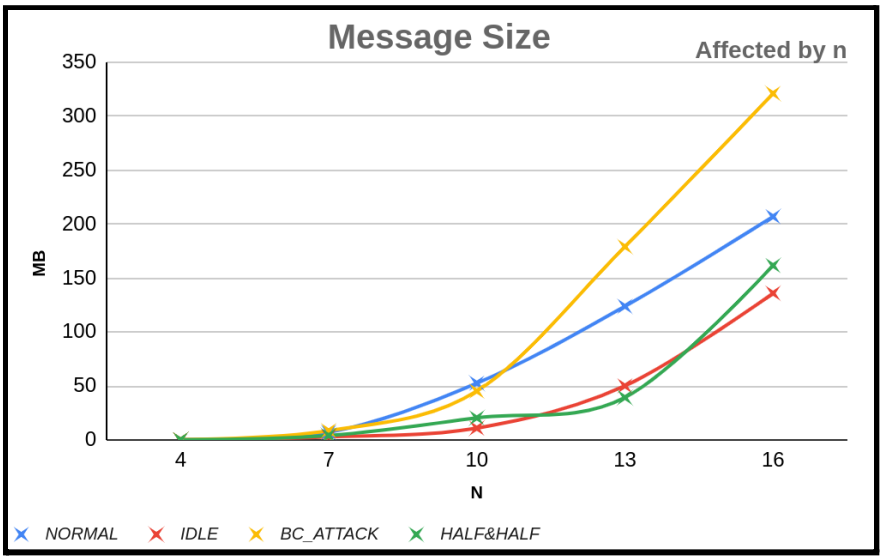

Figure 5.9: Message Size affected by the increment of n

Now, we are moving on to the message complexity and size measurements. Firstly, the Idle scenario acts exactly the same way as in the previous situation as we can see, thus both complexity and size are less than the Normal scenario for the reasons furtherly explained above. Next, by looking at the BC attack scenario's complexity we can understand why the operation latency for 13 and 16 nodes was bigger for this situation, because we can see that the average messages exchanged are much more than in the other scenarios, and consequently the average message size. Additionally, the Half and Half attack scenario has really small average size because half the messages sent are just one (1) byte, and the complexity is less than the normal one, which kind of explains why the latency now is almost equal to the normal and not larger. Finally, the Normal scenario indicates that the theoretical message complexity is just right as its growth is exponential based on n^3, because the nodes are connected with point-to-point links.

5.4. Comparison with Existing Work

Now, let us try to compare our experimental results with the ones of other existing algorithms. Starting with the results presented by the Hashgraph Protocol paper [20], we can notice that they firstly compare the transactions per second each algorithm can serve with four (4) nodes in the cluster and a seven (7) seconds latency. From the experiments we took we can classify our implementation near the HB Protocol. This is more of an assumption, because in order to demonstrate this we need to run the algorithm under the

same resources, programming language and message library as in the paper's [20] experiments, which does not really apply in our case.

Next, they compared only Hashgraph and Honey Badger's latencies, with the results being really interesting, as it presents Hashgraph being ten (10) times faster than HB as the number of servers increases. However, we can only compare our implementation to these results theoretically, due to our lack of resources to test and for the reasons explained in the previous paragraph. Therefore, we can assume that our implementation will be much faster than HB Protocol, as the latency of HB seems to grow rapidly with the increment of the nodes, but in [1] theoretically and as shown in our experiments, our latency will be *O(f)*. However, the latency of Hashgraph Protocol seems to be around two (2) times faster than the algorithm in [1], as Hashgraph seems to not be affected that much by the increment of the nodes, but the *O(f)* time complexity of our implementation is at all times slower.

Protocol	# nodes	tps	latency (sec)
HG	4	22,000	7
HB	4	1,500	7
BEAT0	4	2,000	7
BEAT1	4	600	7
BEAT2	4	700	7

	tps		latency (sec)	
# nodes	HB	HG	HB	HG
32	7,500	7,000	5	5.1
40	10,000	8,000	10	8.4
48	12,000	6,000	48	8.4
56	11,000	6,000	100	10
64	5,000	5,000	200	12.5
104	2,500	2,500	250	25

Figure 5.10: Latency results of existing algorithms from Hashgraph paper [20]

Another paper that illustrated some latency results is BEAT [19], and here we can directly compare with the results we took in our experiments, although the base of the experiments is different, meaning the hardware resources, the programming language and the messaging library. We can see that our results are around one (1) second slower than BEAT's for *f=1* and *f=2*, but as the number of Byzantine nodes in the system increases our latency will grow only a second at a time theoretically, but for these algorithms the increase is more rapid.

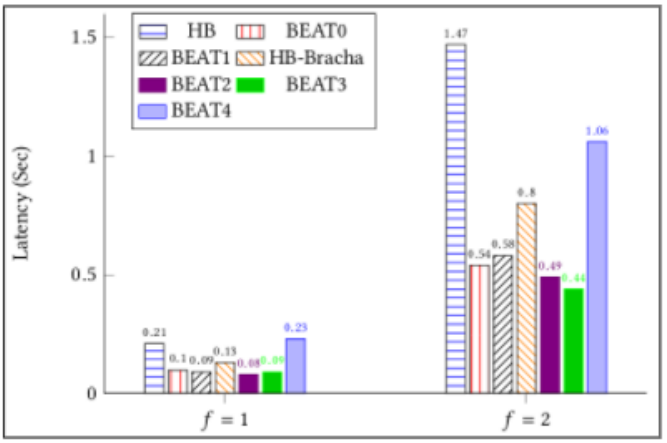

Figure 5.11: Latency results for BEAT from corresponding paper [19]

5.5. Experimental Summary

Overall, we can conclude that while our results are not the best possible, the outcome looks really promising. The algorithm in [1] does not use any digital signing and cryptography in general, except from a signature that verifies that each message is from the right sender, to reduce the large overhead these algorithms come with. However, to achieve this it broadcasts many more messages per round, $O(fn^3)$. Therefore, even though it accomplishes some really good results, an algorithm that uses a smart idea to reduce the amount of messages exchanged, like Hashgraph [20], seems to have better results.

Chapter 6

Conclusion

6.1. Summary

After researching the literature Byzantine Fault Tolerance in general and several BFT algorithms, we decided to use the Go programming language along with the ZeroMQ library as the messaging channel, to implement and evaluate the algorithm of [1]. We designed a set of experiments with the goal of testing the capabilities of the algorithm and our implementation in terms of performance, operation latency and message complexity. Unfortunately, our results were not the best possible due to the hardware resource bottlenecks we faced, but even from these, the outcome is likely promising. In general, the results touch the surface of the theoretical performance evaluation, with operation latency being around f seconds when the machines running as servers have good enough resources, and message complexity at approximately fn^3 messages.

6.2. Challenges

One of the most difficult parts of this project was building the best achievable inter-processor communication, because even though we had as a basis the approach of an older thesis [23], we performed various improvements mainly to achieve asynchrony alongside with the serialization and deserialization of complex structs to a series of bytes, since the API of ZeroMQ [14] supports only the delivery of bytes. Another difficulty was debugging the system, since in a distributed system is much more challenging than debugging a single processor program, let alone debugging a distributed system with

concurrency. Additionally, translating the algorithm from a pseudocode to Go was challenging, since Go [13] does not quite support functional programming, but its advantages were a lot more and thus makes up for all the additional effort we put in. Finally, the fact that we had very limited resources, as the number of machines in the cluster we used and the low specs of the machines, restricted our ability to attain better results. Apart from these challenges, we believe we achieved our main goal, that of having a comparable outcome.

6.3. Retrospection

Looking back we can not really say that a lot of changes would be applied in the methodology we followed throughout both research and implementation phase, except maybe some minor things. The main alternative we would take has to be algorithm related. Even though the algorithm is really good and avoids using digital signatures by any means, and this really helps avoiding the huge overhead signing brings, it requires many more reliably broadcasted messages to achieve this. Therefore, the change would be as suggested in a sentence in the algorithm's paper [1], to replace the second Reliable Broadcast usage in Multi-valued Consensus with a simple broadcast of the message and to make sure everyone gets the message, all non-faulty processes that receive $n-2f$ messages with value v have to resend these messages to all other processes. This optimization reduces both time and message complexities, but we did not have the time to implement it. Furthermore, we would have tried to remotely test our system much earlier to identify the problem with the low resources and try to fix it in time, in order to achieve the best possible results, something we believe we did not accomplish. Therefore, it would be great to run the experiments on an even more geographically distributed network, but also with processors of greater capabilities. Lastly, Go is an excellent language and it has facilitated the development of a distributed multithreaded system by a lot, however, using an object-oriented language with better support for functional programming would have helped in minimizing the size of the code base [24, 25].

Bibliography

[1] Miguel Correia; Nuno Ferreira Neves; and Paulo Veríssimo. *From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures*, Comput. J. 49(1), 82-96, (2006)

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

[3] Achour Mostefaoui; Moumen Hamouma; and Michel Raynal. *Signature-Free Asynchronous Byzantine Consensus with t<n/3 and O(n^2) Messages*, J. ACM 62(4): 31:1-31:21, (2015)

[4] Mehmet Karaata; and Ali Hamdan. *Reliable Channels for Systems in the Presence of Byzantine Faults*, MATEC Web of Conferences. Vol. 42. EDP Sciences, (2016)

[5] Leslie Lamport; Robert Shostak; and Marshall Pease. *The Byzantine Generals Problem*, ACM Trans. Program. Lang. Syst., 4(3):382-401, (1982)

[6] Kevin Driscoll; Brendan Hall; Håkan Sivencrona; and Phil Zumsteg. *Byzantine Fault Tolerance, from Theory to Reality*, In International Conference on Computer Safety, Reliability, and Security, 2003, pp. 235-248, (2003)

[7] Maarten van Steen; and Andrew S. Tanenbaum. *A brief introduction to distributed systems*, Computing 98(10): 967-1009, (2016)

[8] Leslie Lamport. *Proving the Correctness of Multiprocess Programs*, IEEE Trans. Software Eng. 3(2), pp. 125-143, (1977)

[9] Coulouris George; Jean Dollimore; and Tim Kindberg. *Distributed Systems: Concepts and Design (3.ed.)*, International computer science series, Addison-Wesley-Longman, (2002)

[10] Michael J. Fischer; Nancy A. Lynch; and Michael S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process*, J. ACM 32(2), pp. 374-382, (1985)

[11] Fred B. Schneider. *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*, ACM Comput. Surv. 22(4), pp. 299-319, (1990)

[12] Zibin Zheng; Shaoan Xie; Hongning Dai; Xiangping Chen; and Huaimin Wang. *An Overview of Blockchain Technology:Architecture, Consensus, and Future Trends*, BigData Congress 2017, pp. 557-564, (2017)

[13] The Go Programming Language. *https://golang.org/*, (Accessed: 2021-05-24)

[14] ZeroMQ. https://zeromq.org/, (Accessed: 2021-05-24)

[15] iMatix Corporation. *https://github.com/imatix*, (Accessed: 2021-05-24)

[16] ZeroMQ Socket API. *https://zeromq.org/socket-api/*, (Accessed: 2021-05-24)

[17] Miguel Castro; and Barbara Liskov. *Practical Byzantine Fault Tolerance* and *proactive recovery*, ACM Trans. Comput. Syst., 20(4), pp. 398-461, (2002)

[18] Andrew Miller; Yu Xia; and Kyle Croman. *The Honey Badger of BFT Protocols*. CCS 2016: 31-42, (2016)

[19] Sisi Duan; Michael K. Reiter; and Haibin Zhang. *BEAT: Asynchronous BFT Made Practical*, CCS, 2018, pp. 2028-2041, (2018)

[20] Leemon Baird; and Atul Luykx. *The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers*, COINS 2020, pp. 1-7, (2020)

[21] Christian Cachin; and Jonathan A. Poritz. *Secure Intrusion-tolerant Replication on the Internet*, DSN 2002, pp. 167-176, (2002)

[22] Christian Cachin; Klaus Kursawe; and Victor Shoup. *Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography*, J. Cryptol. 18(3): 219-246, (2005)

[23] Nikolas Pafitis. *Implementation of Self-Stabilizing BFT Using Go And ZeroMQ*, Diploma Project, Dept. of Computer Science, University of Cyprus, (2017)

[24] BFTWithoutSignatures. *https://github.com/v-petrou/BFTWithoutSignatures*, (Accessed: 2021-05-24)

[25] BFTWithoutSignatures_Client. *https://github.com/v-petrou/BFTWithoutSignatures_Client*, (Accessed: 2021-05-24)

[26] Cantor's Pairing Function. *https://mathworld.wolfram.com/PairingFunction.html*, (Accessed: 2021-05-24)

[27] Golang Gob. *https://golang.org/pkg/encoding/gob/*, (Accessed: 2021-05-24)