University of Cyprus
Department of Computer Science

MASTER THESIS

Implementation, Validation, and Experimental Evaluation of a Self-stabilizing Byzantine-tolerant Multivalued Consensus Algorithm

Andreas N. Charalampous

Supervisor:

Professor **Chryssis Georgiou**

NICOSIA, CYPRUS

JULY 2022

IMPLEMENTATION, VALIDATION, AND EXPERIMENTAL EVALUATION OF A SELF-STABILIZING BYZANTINE-TOLERANT MULTIVALUED CONSENSUS ALGORITHM

Andreas N. Charalampous

This Dissertation Thesis is submitted for Partial Fulfillment of the requirements for

obtaining the Master's Degree in Computer Science

from the Department of Computer Science of the University of Cyprus

Recommended for Acceptance
from the Department of Computer Science
July 2022

Abstract

Distributed computing has been in the foreground for decades, with distributed systems having numerous applications. Due to their architecture, one fundamental problem that these systems often need to solve is the consensus problem. To accomplish that, the nodes of such a system must cooperate to decide on a value.

A significant percentage of the system nodes should agree on the same value. An impediment to this is that it should expect some of these nodes to act arbitrarily, deviating from their expected behavior, leading to the whole system deciding on an invalid value during consensus. These nodes are called Byzantine or faulty, and their malicious act may happen because of some internal software or hardware malfunction or a malware attack. A fundamental property of distributed systems is that their non-faulty nodes can achieve consensus in the presence of Byzantine nodes.

Another property of a Distributed System is Self Stabilization and how the system behaves to handle errors. Such self-stabilizing systems can automatically recover from arbitrary transient faults, violating the system operation assumptions. Examples of such faults might be simple bit-flips in state variables or messages, but with the code left induct.

In this thesis, a self-stabilizing Multivalued consensus algorithm is implemented, validated, and experimentally evaluated in the presence of up to t Byzantine processes, where $t < n/3$, with n being the total number of processes. Consensus is performed on an asynchronous message-passing network using the Go programming language and the ZeroMQ message library. Experiments are performed on a local workstation and the Emulab testbed platform.

APPROVAL PAGE

Dissertation Thesis

IMPLEMENTATION, VALIDATION, AND EXPERIMENTAL EVALUATION OF A SELF-STABILIZING BYZANTINE-TOLERANT MULTIVALUED CONSENSUS ALGORITHM

Presented by

Andreas N. Charalampous

Supervisor

Dr. Chryssis Georgiou

Committee Member

Dr. Ioannis Marcoullis

Committee Member

Dr. George Pallis

University of Cyprus

July 2022

Acknowledgments

First of all, I would like to express my gratitude to my supervisor, Professor Chryssis Georgiou, for giving me the chance to work on and implement this project. Next, for guiding me through the thesis object by providing his knowledge and experience for the best and most effective outcome.

Moreover, I would like to express gratitude to my fiancé Florentia for always being next to me and helping me throughout my studies and career from the beginning. Kudos for her patience.

Finally, many thanks to my family and friends for their support all these years.

Table of Contents

List of Figures

Abbreviations

MVC: Multivalued Consensus

BC: Binary Consensus

VBB: Validated Byzantine Broadcast

BRB: Byzantine Reliable Broadcast

BVB: Binary Value Broadcast

AWS: Amazon Web Services

Chapter 1

Introduction

1.1 Motivation

Distributed systems have been part of our everyday life since the 1970s. They are used in a significant percentage of the systems we use, with networks, databases, and distributed real-time systems being some of them. A distributed system has components distributed on different computers, all communicating through the network to achieve their common goal [1]. Despite many computers, end-users see and use a distributed system as a standalone interface. A distributed system's computers are usually called nodes, processors, processes, and others. Their main difference from a standard parallel system is that they operate far from each other, even on different continents.

Distributed computing, which studies the principles of distributed systems, depends on their topology, as the nodes act as needed to communicate and solve problems. Distributed systems are expected to contain faulty nodes during operation, which, when low in numbers, should not affect the system's overall behavior. One of the most challenging problems distributed systems must solve while operating and expecting faulty nodes is reaching c*onsensus* [2]. The consensus problem definition is straightforward but more complex to solve in action. The goal is that each non-faulty node supports a value from a predefined set of values, and in the end, they agree on one. In the simplest form, *Binary Consensus* [3], all non-faulty nodes must decide on a value from a set of only two possible values, {0, 1}. The consensus problem is called *Multivalued Consensus* when the set contains more than two possible values [3].

The consensus problem is hard to solve because of faulty nodes, commonly named Byzantine nodes, after the Byzantine Generals Problem [4]. Byzantine nodes fail to follow the algorithm instructions, for example, changing or not sending messages to other nodes. It can result from the node's hardware or software malfunctions or even malware attacks, where an adversary controls the node. It is a vital property of Distributed Systems to defend against Byzantine nodes.

Another type of failure is *arbitrary transient faults* that rarely happen on a node. Transient faults can also be a violation of how the node is designed to work, where this can be a corruption on a control variable, like the program counter or the messages sent or received. Distributed systems can be designed to handle such errors using *Self-Stabilization* [5]; as the name implies, the node handles and bypasses the occurring fault without human intervention.

It is easy to notice that there are many properties that a distributed system must have in order to operate and solve the problems that it was designed for while handling the variety of errors that may occur.

1.2 Objective and Contribution

This thesis aimed to implement, validate and experimentally evaluate the Self-stabilizing Byzantine tolerant Multivalued Consensus algorithm for asynchronous messages passing by Duvignau *et al*. [6]. To the best of our knowledge, it is the first algorithm that solves the Multivalued Consensus problem in asynchronous message-passing that is both Self-stabilizing and Byzantine tolerant, as it can handle up to $t < n/3$ Byzantine nodes. The relevant paper analyzes and theoretically evaluates the algorithm [6]. However, it is essential for the algorithm to be implemented and experimentally evaluated to verify the theoretical specifications and capabilities and confirm that there are no limitations when applied in practice.

The algorithm was studied, implemented, and evaluated on real-world-like simulations for validity. Then it was experimentally compared to a non-self stabilizing Byzantine-tolerant multivalued consensus implementation [7] for performance and message complexity. The algorithm was implemented using the Go Programming Language [8] alongside the ZeroMQ messaging library [9], whereas the experiments were conducted on a local workstation and the Emulab testbed platform [10]. Again, to the best of our knowledge, this is the first implementation, experimental validation, and evaluation of the algorithm in [6].

1.3 Methodology

We used the Go programming language alongside the ZeroMQ library to implement the algorithm. We describe the reasons that led to this decision in Section 2.6.

The overall thesis goal was finished in ten months, as it was done part-time. In order to conduct the Thesis goal, an initial study was performed on the field, containing Distributed Systems, Byzantine Fault Tolerance, Binary/Multi-value Consensus, and Self-stabilization. Additional focus was given to the paper introducing the algorithm [6] and on a thesis that used similar technologies [7].

After the initial study, we got familiar with the Go Programming Language, with which we did not have prior experience, and then with the ZeroMQ message-passing library. We then performed the design phase. We agreed on all application modules, setup, and network configuration so that the application nodes are configured and executed as simply as possible.

With everything ready, we started the implementation of the algorithm from scratch. After implementing the message-passing layer, one by one module, Byzantine Reliable Broadcast (BRB), Validated Byzantine Broadcast (VBB), Binary Consensus (BC), and Multivalued Byzantine-tolerant Consensus (MVC) were built, starting from the bottom of the algorithm stack. At first, we implemented the non-self-stabilizing version for each module and then converted it to self-stabilizing. It was possible to implement the self-stabilizing version of the module directly, but by implementing the simple form, it was helpful to understand each module and its properties. Every module was validated with manual testing, unit, and automation tests that tested whole scenarios when added to the stack.

Finally, after the whole algorithm was finished and tested for validity, it was benchmarked on a local personal computer and then on a cluster of 10 nodes, with 200 CPUs in total, using the Emulab testbed platform [10]. We analyze benchmark results and information in Chapter 6 - Experimental Analysis. Extended timesheet details are shown in Figure 1.

1.4 Document Structure

In Chapter 2 - Background and Related Work, there are references to the Related Work and Background. Specifically, we mention existing Byzantine fault-tolerant algorithms, Multivalued Consensus, and Self-stabilization. Moreover, some introduction to the Go programming language and the ZeroMQ library used for implementing the algorithm, and the Emulab testbed platform for executing experiments.

Chapter 3 - The Algorithm contains a complete explanation of the implemented algorithm with all of its building blocks. Then, in Chapter 4 - Implementation Details, there is a complete description of the implementation details of our solution, and in Chapter 5 - Self-stabilization and Byzantine Fault Tolerance, we show all the covered and evaluated use cases alongside the validation and tests performed. Chapter 6 - Experimental Analysis shows the experimental setup and analysis performed alongside the performance and complexity comparisons to other related work.

Finally, Chapter 7 - Conclusion contains a retrospective where we show all our conclusions and final points related to the implementation and possible future work.

	2021				2022					
	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	June
Initial Meetings (15/09/2021 - 16/09/2021)	🟨									
Meeting for subject and related work										
Study of related work (17/09/2021 - 31/01/2022)	🟦	🟦	🟦	🟦	🟦					
Byzantine Fault Tolerance										
Binary and Multi-value Consensus										
Related thesis and algorithms										
Paper about the algorithm to be implemented										
Technical Training (18/01/2022 - 12/02/2022)					🟧	🟧				
Learning Go programming language										
Learning ZeroMQ library										
Design Phase (13/02/2022 - 20/02/2022)						🟩				
Application and network setup										
Implementation of algorithm (21/02/2022 - 09/04/2022)						🟦	🟦	🟦		
Non-self-stabilizing implemention										
Conversion to self-stabilizing										
Manual testing										
Unit tests - automation tests										
Benchmarking (10/04/2022 - 30/06/2022)								🟫	🟫	🟫
Local Benchmarking										
Benchmarking on Emulab										

Figure 1: Gantt Diagram

Chapter 2

Background and Related Work

2.1 Fault Tolerance

2.1.1 General

Fault tolerance is, if not the most, one of the most critical capabilities of a distributed system. As the name implies, a distributed system must be able to overcome internal partial faults and keep functioning, masking the error as if it never happened, with only some graceful drawbacks in performance. A system is fault-tolerant when the following requirements are met [11]:

1. Availability: the system is available to be used as expected at any time.
2. Reliability: the system can work correctly for an extended period without failure.
3. Security: the system does not allow unauthorized access.
4. Safety: in case the system cannot carry out a failure, leading to the working incorrectly for some time, but with no catastrophic results.
5. Maintainability: system failures can be observed and fixed mechanically.

2.1.2 Types of Faults

Different kinds of faults can occur on a distributed system. Some types are less crucial, which are easier to notice and handle, and other types can be disastrous for a system and sometimes cannot be fixed by the system. Next, we describe the main fault types in distributed systems, starting from the less serious to the most severe [12].

The first type of fault is *crash faults*, which can contain simple system component crashes like a processor crash or a link crash. In this case, a component may stop working without any warning.

The next type of fault is *omission fault*, during which a component may omit the execution of a specific operation. For example, during communication omission, a node does not send a message that was supposed to be sent. Another common type of fault is the *timing fault*, which can happen when a component fails to execute an operation during a sufficient time window.

The final and most crucial type of fault are *Byzantine faults*, which are the most difficult to handle and can even create issues that cannot be solved. These errors are considered arbitrary or malicious and happen when a faulty component has an arbitrary behavior. For example, a node can send invalid messages, pretending to be another processor, and stay idle while executing. These are the kind of faults that we study below.

2.1.3 Byzantine Fault Tolerance

Since Byzantine Faults are the most severe fault types, a distributed system must be *Byzantine fault-tolerant*. The name Byzantine comes from the Byzantine Generals Problem introduced by Leslie Lamport, Robert Shostak, and Marshall Pease [4].

The problem definition states that several Byzantine Generals, each with their army division, are camping around an enemy city, planning to attack. For their attack to be successful, they must coordinate to simultaneously attack all or at least a significant number of divisions. This operation is also known as reaching *consensus*. Generals can only communicate through messengers, saying if they should attack or retreat, and what makes the operation complicated is that a general or generals are traitors. In Figure 2, we see an illustration of the Byzantine Generals Problem, where on the top image is shown what each general said if he would attack or retreat and on the bottom what exactly he did. Loyal generals assumed that everyone would attack, while traitors, in the end, retreated. Thus, the attack was unsuccessful.

Through this problem, Leslie Lamport, Robert Shostak, and Marshall Pease proved that in the presence of Byzantine Generals, consensus could not be reached if the number of Byzantines was considerable. Specifically, when there are *n* generals total, the number of Byzantines *t* must not exceed *(n/3) – 1*. The same applies to distributed systems, even if the system is synchronous, with a guaranteed common global notion of time and operations taking place in synchrony.

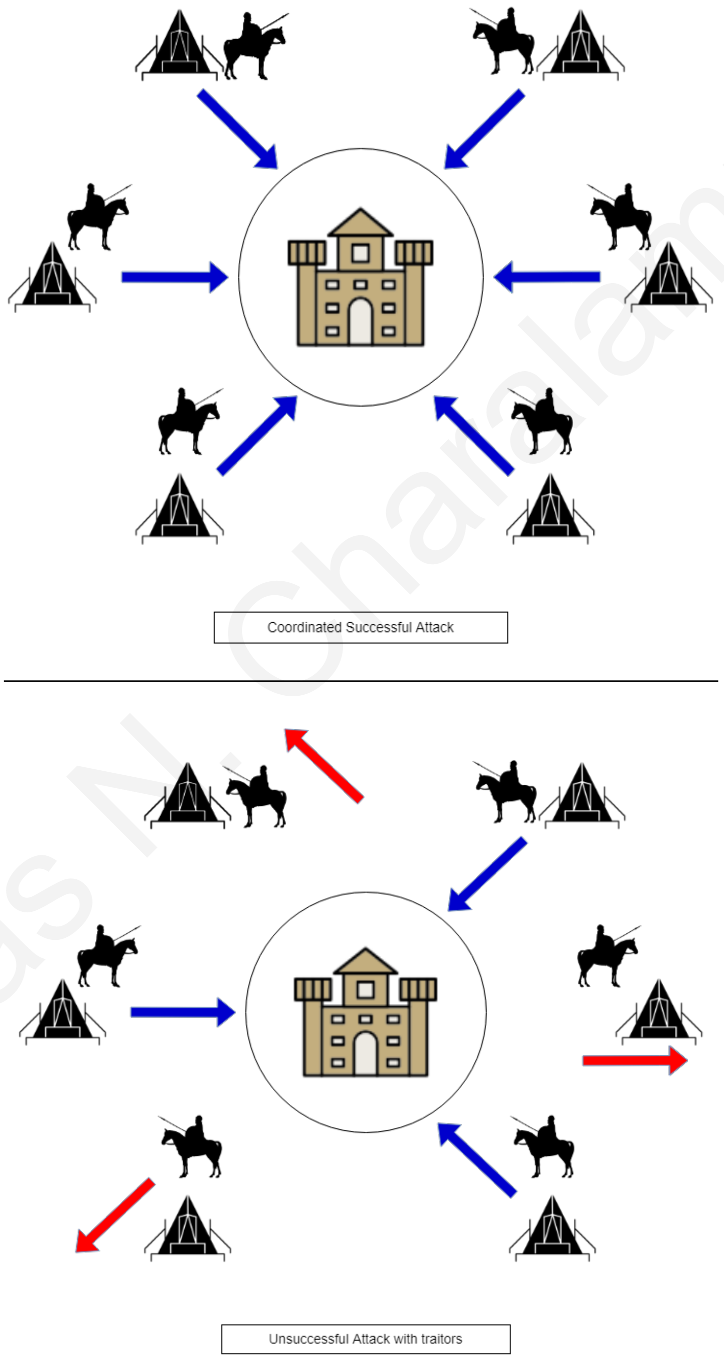
Coordinated Successful Attack
Unsuccessful Attack with traitors

Figure 2: Illustration of the Byzantine Generals' Problem

2.2 Consensus

As mentioned in Section 2.1.3, when we have a decentralized system whose components need to coordinate and agree on an action or value, that is called consensus [3]. In distributed systems, nodes are constantly called to reach consensus to function correctly. It was also made clear that the consensus problem becomes much more complicated and even unsolvable in the presence of Byzantine nodes. In its simplest form, consensus must satisfy the following three (3) requirements:

1. <u>Consistency:</u> All correct nodes agree on the same value.
2. <u>Validity:</u> The decided value was initially proposed from at least a correct node.
3. <u>Termination:</u> Every correct node eventually decides on a value.

2.3 FLP Impossibility Result

After looking at the Fault Tolerance and Consensus properties, we will see why their properties can not be simultaneously assured in asynchronous systems without some additional mechanism. We saw above that the nodes of a distributed system communicate and exchange messages to achieve consensus. In synchronous systems, faster nodes eventually wait for messages from the slowest nodes. However, in asynchronous systems, there is no specific limit that it will take for a slow node to respond. Because of this, we cannot be sure if a node faulted or is simply slow.

The FLP Impossibility result, which took its name from the authors that introduced it, **Fischer**, **Lynch,** and **Paterson** [13], states that in asynchronous distributed systems, if a single failure occurs, then the system cannot reach a consensus (agreement-safety / termination-liveness). Therefore the **agreement**, **fault tolerance**, and **termination** properties cannot be satisfied simultaneously in asynchronous systems.

There were multiple approaches to solve the FLP Impossibility result, in which specific mechanisms are used. One of them is adding synchrony assumptions in the asynchronous system, for example, using a predefined network delay. Other options include failure detection mechanisms [14] for detecting faults or non-deterministic models with randomization [15].

2.4 Self-Stabilization

In the previous sections, we listed the different types of faults and why fault tolerance is necessary for a distributed system. A fault-tolerant mechanism is explicitly designed depending on assumptions that can be variables or protocols, which help distributed systems defend against faults. This sub-chapter views another type of fault, arbitrary-transient-faults, and how a system can detect and overcome them using self-stabilization.

2.4.1 Arbitrary transient faults

We said that fault tolerance depends on critical system assumptions. It is fair to characterize these assumptions as necessary since one single violation can break fault tolerance and even be fatal to the system. Another type of fault that can affect a distributed system is *arbitrary-transient-faults* [16], which temporarily violate the assumptions that a system or network was designed to follow. They are minimal in terms of how they violate the assumptions. However, as already said, they can even be lethal for the distributed system, turning the system completely useless, making human intervention necessary for stabilizing it again.

This kind of error rarely happens, in an unexpected way, making it impossible to detect it at the time that it happens. It can be tiny and simple, like a single bit-flip on a variable or a message. Such small alternation can happen on the program counter, bringing the system to a faulty state. The system can not recover since it was never designed to be in that state [17]; for example, on a blocking message receive operation, without performing a send operation first.

2.4.2 Self-stabilization after arbitrary transient faults

Self-stabilization [18] is precisely how a system can detect and recover when an arbitrary transient fault happens. As the name implies, when designed with self-stabilization properties, a system that comes in an arbitrary state can stabilize and correct itself without human interaction and return to a valid operating state.

The self-stabilization notion was first mentioned by Edsger Dijkstra while solving the mutual exclusion problem [18]. Dijkstra stated that when there is a violation in the assumptions that a system follows in order to operate and the system goes into an arbitrary state, then with self-stabilization, the system can recover.

Based on this, a distributed system needs two properties: (1) an initial (reset) state and (2) the ability to recover to that state from every arbitrary state. An example of an initial reset state could be the program counter pointing at the beginning of a distributed algorithm and resetting the control variables related to the algorithm. For the latter property, a system performs checks at specific points in the algorithm, and in case a violation is detected, it returns to the initial state.

Even though self-stabilization is considered a fault tolerance mechanism, it should be distinguishable from other fault tolerance methods. As mentioned in the previous sections, fault tolerance methods mask errors and prevent failures. In contrast, on the other side, self-stabilization guarantees recovery after a transient failure occurs. The transient failure may have a noticeable effect on the system operation before it recovers.

2.5 Existing Algorithms

As mentioned by Duvignau *et al.* [6], the studied algorithm, by the best of their knowledge, is the first self-stabilizing, Byzantine, and intrusion-tolerant algorithm for solving multivalued consensus in asynchronous message-passing systems. Existing solutions consist of non-Byzantine fault-tolerant solutions that do not use self-stabilization [19] and self-stabilizing solutions that do not have fault-tolerance against Byzantine failures [20] [5] [21] [22].

The first approach of Byzantine tolerance in asynchronous systems reducing multivalued consensus to binary consensus was that of Ben-Or, Kelmer, and Rabin [23]. However, it did not consider intrusion tolerance which later Mostéfaoui and Raynal [24] [25] and Correia, Neves, and Veríssimo [26] [27] proposed. We will see later that the MVC-no-intrusion (intrusion tolerance) requirement states that the decided value cannot be a value proposed only by Byzantine nodes.

A related solution for solving Binary Consensus is that of Mostéfaoui et al. [28] [29], who presented an asynchronous randomized solution with common coins. Georgiou et al. [16] proposed a self-stabilizing variation on Mostéfaoui et al.'s algorithm. The self-stabilizing Byzantine-tolerant Binary consensus object from this proposal will be used in the studied algorithm by Duvignau, Schiller, and Raynal [6], from which randomization is inherited therefore bypassing the FLP Impossibility result.

2.6 Go Programming Language

Go, or Golang [8], is a relatively new open-source programming language, with its first version being released in March of 2012. It was designed and implemented at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Its main goal was to improve programming productivity, providing multithreading and network tools imported at the syntax level. Designers shared a common opinion against languages used at Google, increasing the complexity of Google's codebase. However, they focused on designing Go, keeping the valuable capabilities of each one of them. This approach gave Go the characteristics of a Static typing, compiled, run-time efficient language like C. Easily readable and usable like Python and Javascript, and High-performance networking and multiprocessing [30]. Another important feature of Go is that it contains goroutines and channels in the language syntax. The first is used to implement multithreading applications, and the second is to provide communication mechanisms between threads. These features make implementing, reading, and maintaining multithreading applications easy.

The mentioned characteristics make the language one of the simplest for server-side programming, game development, cloud-based programming, and Data-Science [31]. Go offers easy-to-read documentation about its features and packages directly through the official website [8]. Combined with the increasing community and support, the language is considered one of the most hyped and loved programming languages.

The combination of the above led us to use Go to implement the algorithm, as it perfectly fits our needs. Also, the existing implementation of a related algorithm was conducted with Go, so we can be more precise in comparing the two. Initially, we started the implementation with *go1.17* and upgraded to *g1.18* shortly after to get advantages of added features like Generics.

2.7 Emulab – Cloudlab

Our implementation was evaluated and benchmarked using the Emulab platform technologies. Emulab [10] is a network testbed that allows researchers to create environments, setups, and configurations for general development, testing, debugging and evaluating experiments and systems. It provides many physical and visual nodes with various specs so that users can build the most suitable environment for their experiments.

Researchers can request access to the platform by filling out an application form, and after their application is evaluated, they are granted access. After that, they are free to allocate resources, depending on availability, to set up clusters for running experiments. We managed to get access by creating an account using our Institutional Email and specifying the purpose of our experiments. Instructions for getting access and documentation for using the platform are included in the Emulab manual [32].

For our experiments, we used Cloudlab [33], which uses the same technologies, interface, and user accounts as Emulab but provides far more resources, including access to the Emulab resources. During the time we used it, the Cloudlab deployment consisted of more than 25,000 cores distributed across three sites at the University of Wisconsin, Clemson University, and the University of Utah [33].

2.8 ZeroMQ

ZeroMQ, also known as ØMQ, 0MQ, or ZMQ [9], is a high-performance asynchronous network message-passing library. It is designed mainly for distributed and concurrent systems. The Zero prefix describes the framework's profile, referring to its minimalism, with zero brokers, high performing with zero latency, zero cost as it is free, and zero administration. On most operating systems, it provides APIs for the most known programming languages, like C, C++, Java, Go, Python, Rust, and many other programming languages. With these asynchronous features, the library was the best fit for our algorithm implementation [9].

2.8.1 ZeroMQ Sockets and Patterns

The library provides a plethora of communication sockets and patterns, giving developers the flexibility to design and structure their network however they want. There are many types of sockets provided that can be combined to satisfy every scenario. Next, we describe some of them.

2.8.1.1 REQ and REP sockets

Two of the most basic sockets are REQ and REP, which stand for request and reply, respectively; combined, they give a basic client-server pattern. These two sockets are synchronous, limiting their practical, real-world applications. In order to function, these two ports must send and receive messages alternately. For example, they cannot send two messages consecutively without receiving a message between them. The REQ socket first sends then receives in a Send, Receive, Send Receive pattern, whereas

REP first receives, then sends in a Receive, Send, Receive, Send pattern. Figure 3 shows the primary valid and invalid use of REQ and REP sockets.

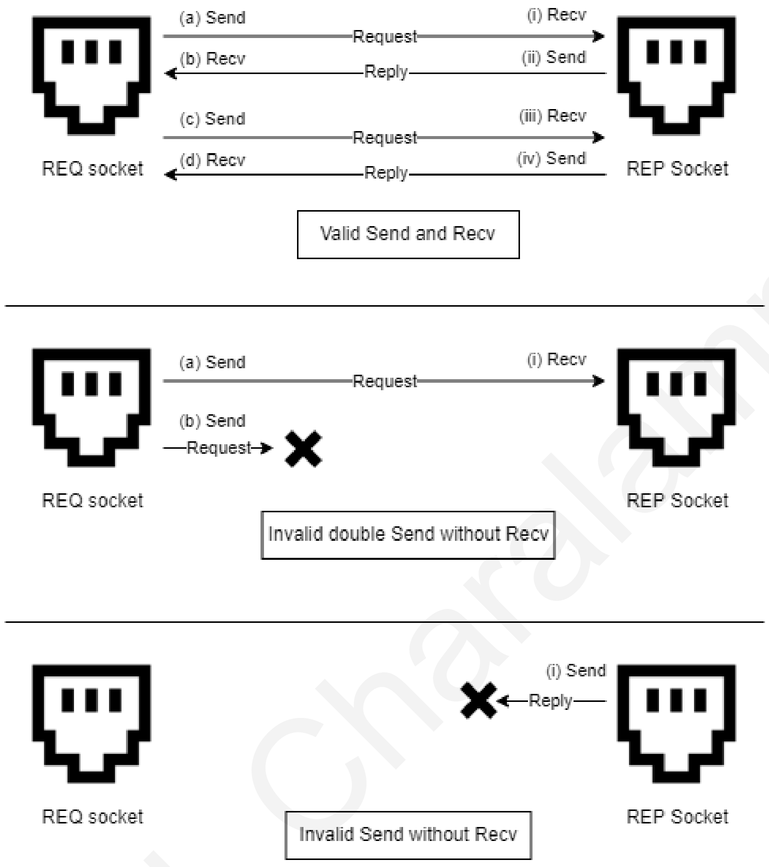

Figure 3: Valid and invalid scenarios of REQ and REP sockets

2.8.1.2 DEALER and ROUTER sockets

DEALER and ROUTER sockets are considered the non-blocking, asynchronous variant of REQ-REP sockets, DEALER (old name: XREQ) acting REQ, and ROUTER (old name: XREP) as REP sockets. The most important thing that allows ROUTER to work asynchronously is that it expects all incoming messages to contain a leading identity frame containing information about its sender. Neither port has any restriction on the sending/receiving sequence pattern.

2.8.1.3 PUB and SUB sockets

The pair of PUB, meaning publisher, and SUB, meaning subscriber, are used for the well-known publisher-subscriber pattern. PUB sockets publish messages, and SUB sockets receive those messages. PUB sockets cannot receive, and SUB sockets cannot send messages on the other side. Since SUB sockets do not send any response to acknowledgments to PUB messages, their communication is asynchronous.

2.8.1.4 PUSH and PULL sockets

ZeroMQ sockets can also apply a pipeline pattern in cases where few nodes push work to many workers, who then forward results to others. A PUSH socket communicates with several PULL peers and can only send messages and not receive. Similarly, a PULL socket is connected to some PUSH socket peers and is allowed to receive and not send messages. A property of this setup is that PUSH and PULL sockets do not know anything about their peers.

2.8.1.5 PAIR sockets

Pair sockets are used in specific scenarios and are unsuitable for TCP network communication. They are mainly used for inter-thread communication within a single process and can only connect to a single peer at a time.

Chapter 3

The Algorithm

3.1 Algorithm Structure

Like almost every consensus algorithm (or module), the studied Multivalued Consensus algorithm is built on top of other algorithms. As shown in Figure 4, Duvignau, Shiller, and Raynal's [6] whole structure contained a multivalued Byzantine-tolerant consensus algorithm (MVC) built on top of a Binary Consensus Object (BC) and Validated Byzantine Broadcast (VBB). VBB is built on top of Byzantine Reliable Broadcast (BRB). On top of the multivalued consensus algorithm, a reliable broadcast with a total-order delivery algorithm can then rely on it. On top of that, state-emulation is achieved. All the above depend on an asynchronous message-passing system. The studied algorithm of Duvignau, Shiller, and Raynal contains only MVC, VBB, and BRB, assuming the existence of a BC object is shown in blue cells in Figure 4. Our implemented and used (Byzantine-tolerant binary consensus [16]) protocols are shown in red-bordered cells.

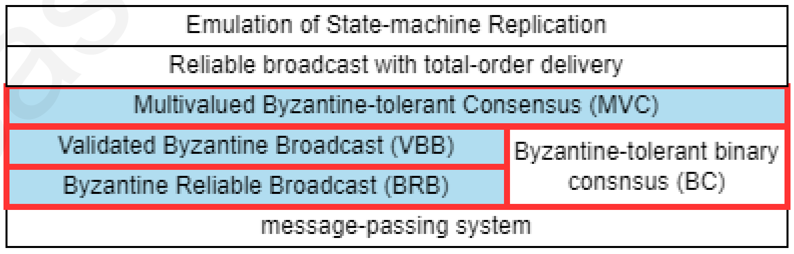
Emulation of State-machine Replication
Reliable broadcast with total-order delivery
Multivalued Byzantine-tolerant Consensus (MVC)
Validated Byzantine Broadcast (VBB)
Byzantine Reliable Broadcast (BRB)
Byzantine-tolerant binary consnsus (BC)
message-passing system

Figure 4: The studied architecture. Algorithms studied by Duvignau, Schiller, and Raynal are in blue cells. Our implementation is shown in red-bordered cells.

The following sections show the studied protocols starting from the protocol stack's base and moving to the top in their non-self-stabilizing version. Later, we show how they are converted to self-stabilizing. All the shown algorithm figures in the section are written in simplified pseudocode syntax and originated from Duvignau, Shiller, and Raynal's paper [6].

3.2 Protocol Stack

3.2.1 Message-passing system

Starting from the base, a vital property for a distributed algorithm is communication and networking. In the basic non-self-stabilizing form, the broadcasting algorithms depend on reliable communication channels. Such channels offer basic guarantees; for example, all sent messages are eventually delivered (**fairness**), a received message is created and sent from some process (**no-creation**), and every sent message is received precisely once (**no-duplication**) [34]. We will discuss later further how these properties are satisfied in our implementation.

3.2.2 Byzantine Reliable Broadcast

The first broadcasting algorithm is Byzantine Reliable Broadcast, proposed by Bracha and Toueg [35]. The BRB algorithm has the abstraction of two (2) primary operations that of **brbBroadcast(message)** in order to broadcast a value to all other peers and **brbDeliver()** raised from a node that received a message from another node so that the following assumptions are applied:

1. BRB-validity: if a correct node raised *brbDeliver* for a message *m* from a process *p*, process *p* invoked *brbBroadcast(m)*.
2. BRB-integrity: a correct node cannot *brbDeliver* the same message from the same process more than once.
3. BRB-no-duplicity: two different correct processes cannot *brbDeliver* different messages from a process *p*, where *p* can even be faulty.
4. BRB-Completion-1: when a correct process *p* invokes *brbBroadcast(m)*, all correct nodes *brbDeliver* its message *m*.

5. <u>BRB-Completion-2:</u> if a correct process p_i *brbDeliver* a message *m* from p_j that can even be faulty, then all correct nodes eventually *brbDeliver* message *m* from p_j.

Each message broadcasted with BRB can be one of three (3) types; **INIT** when a process broadcasts its initial value, **ECHO** for messages containing the value of another node, and **READY** when a node is ready to *brbDeliver* a message from a node. BRB can guarantee reliable broadcast assuming $t < n/3$, where *t* is the number of faulty processes.

3.2.2.1 No-Duplicity Broadcast

BRB depends on a simpler broadcast algorithm, No-Duplicity Broadcast (ND-Broadcast), proposed again by Toueg [15]. ND-Broadcast, similarly to BRB, offers two (2) operations; *ndBroadcast(message),* which is the same as *brbBroadacast(message),* and *ndDeliver()*, which is raised before *brbDeliver()*.

The ND-Broadcast algorithm is shown in Figure 5. Assuming that we have a process p_i that *ndBroadcasts* a message m_i and every other process receives this message, p_i invokes *ndBroadcast*(m_i), with *INIT*($_i$, m_i) being sent to every other process. When another process p_j receives *INIT*($_i$, m_i) for the first time, it broadcasts an *ECHO*($_i$, m_i) to the rest of the processes. If p_j then receives *ECHO*($_i$, m_i) from at least $(n + t)/2$ nodes, where *n* and *t* are respectively the numbers of all processes and faulty processes, then p_j raises *ndDeliver* event for message m_i from process p_i.

```
ndDeliver(k, mJ):
    // event raised when mJ ND-Delivered from node k

ndBroadcast(m):
    broadcast ND_INIT(m)

on arrival of ND_INIT(mJ) from node j:
    broadcast ND_ECHO(j, mJ)

on arrival of ND_ECHO(k, mJ) from j:
    if received ND_ECHO(k, mJ) from at least (n+2)/2 nodes:
        if not yet invoked ndDeliver(k, mJ):
            ndDeliver(k, mJ)
```

Figure 5: The ND-broadcast algorithm

3.2.2.2 Byzantine Reliable Broadcast on ND-Broadcast

As already mentioned, BRB is built on ND-Broadcast, and actually, it is a continuation of it. The main difference is on the part where *ndDeliver* is raised, where BRB sends the last type of READY message waiting for equivalent messages from other nodes, which guarantees the reliability of broadcasting in the presence of Byzantine processes. The BRB algorithm is shown in Figure 6.

Continuing from the previous algorithm, when a node p_j receives *ECHO*$(_i, m_i)$ from at least $(n + t)/2$ nodes, then and if it did not do it already, it broadcasts *READY*$(_i, m_i)$. Next, when node p_j receives *READY*$(_i, m_i)$ from another node, it performs two checks. Firstly, if it received the same *READY* message from $t+1$ nodes, meaning that at least one correct node sent a *READY*, and if not yet broadcasted, it broadcasts *READY*$(_i, m_i)$. This case can happen in the case where the node receives enough READY messages before receiving enough ECHOs.

Secondly, it checks if it received the same *READY* message from at least $2t+1$ nodes, and if not already done, it raises a *brbDeliver* for that message. This check ensures no two correct nodes *brbDeliver* different values, as *BRB-Completion-2* states.

```
brbDeliver(k, mJ):
    // event raised when mJ BRB-Delivered from node k

brbBroadcast(m):
    broadcast BRB_INIT(m)

on arrival of BBR_INIT(mJ) from node j:
    broadcast BRB_ECHO(j, mJ)

on arrival of BRB_ECHO(k, mJ) from j:
    if received BRB_ECHO(k, mJ) from at least (n+2)/2 nodes:
        if not yet broadcasted BRB_READY(k, mJ)
            broadcast BRB_READY(k, mJ)

on arrival of BRB_ECHO(k, mJ) from j:
    if received BRB_READY(k, mJ) from (t+1) nodes:
        if not yet broadcasted BRB_READY(k, mJ)
            broadcast BRB_READY(k, mJ)
    if received BRB_READY(k, mJ) from (2t+1) nodes:
        if not yet invoked brbDeliver(k, mJ):
            brbDeliver(k, mJ)
```

Figure 6: The BRB-broadcast algorithm

3.2.3 Validated Byzantine Broadcast

The final studied broadcast algorithm is that of Validated Byzantine Broadcast (VBB) and is built on top of BRB. Again, it offers two operations, **vbbBroadcast(message)** for VBB broadcasting a message and **vbbDeliver()** raised when a VBB message is delivered. As the name implies, a message is validated by checking if it is VBB delivered by a number of nodes. When a message from a sender cannot be validated, then a transient error symbol Ψ indicates the invalidity of a message. The VBB requirements are:

1. VBB-validity:
    a. VBB-justification: if a correct node raises *vbbDeliver* for message *m* ≠ Ψ, there is at least one correct node that invoked *vbbBroadcast(m)*.
    b. VBB-obligation: if all correct nodes invoked *vbbBroadcast(m)* for the same message *m*, all correct nodes raise *vbbDeliver* for every *m* from each node broadcasted.

2. VBB-uniformity: if a correct node raises *vbbDeliver* for *m'* ∈ {*m*, Ψ} from a node *p* that can even be faulty, then every other correct node raises *vbbDeliver* for *m'* from *p*.
3. VBB-Completion: when a correct node *p* invokes *vbbBroadcast(m)*, all correct nodes raise *vbbDeliver* for *m* from *p*.

Figure 7 shows the VBB algorithm. The multiset **rec** is used for the implementation, containing all the BRB-delivered values. Also, **equal(v, rec)** returns the number of elements in *rec* equal to *v*, and **differ(v, rec)** returns the number of elements not equal to *v*.

In contrast to the BRB algorithm, where we could assume that only one process broadcasts its value, VBB works with all processes broadcasting their value. There are two (2) types of VBB messages, INIT and VALID, where the first is used mainly for broadcasting the value and the latter for the messages validation procedure. In general, the algorithm is split into two parts. The first part contains broadcasting of values, gathering values of other processes, and validating them. The second part waits for validations from other processes to finally raise *vbbDeliver*.

Initially, every node BRB-Broadcasts a VBB-INIT message *m* and waits until at least *(n-t)* VBB-INIT messages are BRB-Delivered. Those values, as mentioned above, are stored in the *rec* multiset. When that number of messages are BRB-delivered, the node counts how many times he can find his proposed value in *rec* multiset, and if they are more than *n-2t*, then it *brbBroadcasts* a VBB-VALID message with the value *true*, otherwise with the value *false*.

After sending the VBB-VALID message, the node executes a background task for each one of the other nodes, for *vbbDelivering* their value or the error value Ψ. Taking into consideration the background task where node p_i validates the value of p_j, firstly, node *pi* waits for both VBB-INIT and VBB-VALID messages to be BRB-delivered. When both messages are BRB-delivered, the VBB-VALID message value from *pj* is checked. If it is *true*, p_i waits until the VBB-INIT message value of p_j is BRB-delivered from at least *n-2t* nodes. If this happens, p_i VBB-delivers value v_j from p_j. Else if the value is *false*, p_i waits for *t+1* values that are different from p_j to be BRB-delivered. If that happens, p_i VBB-delivers the transient symbol Ψ from p_j.

```
variables:
    rec // multiset that contains all BRB-delivered values
equal(v):
    return number of elements in rec equal to v
differ(v):
    return number of elements in rec not equal to v


vbbDeliver(j, d):
    // raised when the value d is VBB-Delivered from node j

vbbBroadcast(v):
    brbBroadcast(VBB_INIT(i, v))
    wait until len(rec) >= n-t
    brbBroadcast(VBB_VALID(i, equal(v) >= n-2t))

background tasks (one for every node j, j != i):
    wait VBB_VALID(j, x) and VBB_INIT(j, v) BRB-delivered from j
    if x == true:
        wait until equal(v) >= n-2t
        d = v
    else:
        wait until differ(v) >= t+1
        d = Ψ
    vbbDeliver(j, d)
```
Figure 7: The VBB-broadcast Algorithm for node i

3.2.4 Randomized Byzantine-tolerant Binary Consensus

One fundamental property of the studied algorithm is the reduction from Multivalued Consensus to Binary Consensus. The latter is a simplified version of the first one since in MVC, we can have whatever value proposed, whereas, in BC, only two (2) different values can be the outcome of consensus, that of 0 or 1. Binary consensus offers two operations, **binPropose()**, where one of two possible values can be proposed for consensus, and **binResult()**, which returns the result of consensus. There are a set of properties held during Binary Consensus:

1. BC-Validity: if every correct node proposes a value *b*, then every correct node can decide only that value *b*.

2. BC-Agreement: two correct nodes cannot decide on different values.
3. BC-Termination: every correct node eventually decides.

Binary consensus is not contained in the studied architecture of Duvignau, Shiller, and Raynal. For our implementation, we used a Randomized Binary Consensus built on Binary Value broadcast (BVB), shown in Figure 8. The BVB-broadcast algorithm is a basic algorithm in which a node broadcasts its binary value and waits for the values of other nodes. When it receives a value from at least *t+1* processes, and if not done yet, it broadcasts it to the other nodes. If the same value is received from *2t+1* processes, it is added to a set called **bin_values**. We will see below how this set is used during Binary Consensus. The two (2) types of Binary Consensus messages are EST and AUX, sent in different phases.

```
variables:
    bin_values // set that contains binary values received
               // from 2t+1 different processes

bvbBroadcast(b):
    broadcast b

on arrival of value b:
    if received b from at least (t+1) nodes:
        if not already invoked bvbBroadcast(b):
            bvbBroadcast(b)
    if received b from at least (2t+1) nodes:
        add b to bin_values
```
Figure 8: The BVB-broadcast algorithm

The Binary Consensus is split into three (3) phases, executed in rounds. In the first phase, all nodes exchange their proposed binary value by sending it through BVB alongside the current round and the EST tag. Then they wait until their *bin_values* set is filled with at least a value.

In the second phase, nodes send a random value from the *bin_values* set with the round number and the AUX tag. Again they wait for *n-t* AUX messages delivered containing a value that exists in the *bin_values* set.

By having such a value, the node proceeds to the third phase, where it first flips a coin common for all nodes. For the common coin, we used a very dummy generator, which considers the round in which is flipped and a prime number. After getting the common coin value, it checks if the value from the second phase is the same as its round estimation and the value from the common coin. If they all match, the node decides on value *v*. Otherwise, it moves to a new round. Also, if the second phase value does not match the node's round estimation, it adopts that value and moves to the next round. This procedure is repeated until consensus is reached. The complete algorithm is shown below.

```
binPropose(v):
    estimation = v
    round = 0
    do forever:
        round++

        // phase 1
        m = (round, estimation, BC_EST)
        bvbBroadcast(m)
        wait until bin_values not empty

        // phase 2
        aux_value = bin_values[0]
        m = (round, aux_value, BC_AUX)
        broadcast m
        // received values are kept in values set
        wait until n-t messages received
          from different nodes with values
          contained in bin_values

        // phase 3
        common_coin_value = common_coin()
        if set values contains exactly v:
            if v equals common_coin_value:
                return v
            else
                estimation = v
        else
            estimation = common_coin_value
```

Figure 9: The BC Algorithm

3.2.5 Multivalued Byzantine-tolerant Consensus

Next, we look at the goal algorithm, Multivalued Byzantine-tolerant Consensus. MVC contains the **propose()** operation, where a node can propose its value to all other nodes to initialize the consensus procedure. The algorithm depends upon VBB-broadcast communication abstraction and a Byzantine fault-tolerant Binary consensus. The MVC algorithm contains the following requirements:

1. MVC-Completion: all correct nodes eventually decide on a value.
2. MVC-Agreement: two correct nodes cannot decide on different values**.**
3. MVC-Validity: only a value that was proposed can be decided.
4. MVC-no-intrusion: the decided value cannot be a value that was proposed by faulty processes only.

All correct nodes are expected to invoke the *propose()* operation during MVC. There is only one (1) MVC message type: EST. The algorithm is shown in Figure 10.

The first step of a node *i* is that of *vbbBroadcasting* an EST message containing the proposed value. Node *i* waits for EST messages to be *vbbDelivered* from at least *n-t* different nodes, which are kept in a multiset *rec*. When this happens, node *i* tests whether *rec* includes at least *n-2t* non-Ψ replies and exactly one non-Ψ value. The next step contains the part where MVC is reduced to BC since the test result, *true* or *false*, is *binProposed* to BC.

Once the BC result is available, if its value is *false*, node *i* returns the transient error Ψ. If the result value is *true*, node *i* waits until it receives *n-2t* messages from different nodes that match his value, and if this happens, it returns his proposed value *v*.

```
variables:
    rec // multiset that contains all VBB-delivered values
distinct():
    return number of distinct non-psi values in rec
nonPsi():
    return number of non-psi values in rec

bp():
    return nonPsi() >= n-2t && distinct(rec) == 1

propose(v):
    vbbBroadcast(EST(v))
    wait until len(rec) >= n-t
    if binPropose(bp()):
        wait until value v≠Ψ VBB-delivered from n-2t nodes
        return v
    else:
        return Ψ
```

Figure 10: The MVC algorithm

3.3 Self-stabilizing Byzantine-tolerant Multivalued Consensus

Having a complete check on the non-self-stabilizing version of all algorithms, we will address the challenges towards self-stabilization mentioned in Duvignau, Schiller, and Raynal's proposal. We will then list all solutions and show how each algorithm is converted to the self-stabilizing version to give us the Self-stabilizing Byzantine-tolerant Multivalued Consensus.

3.3.1 Challenges and Solutions

3.3.1.1 Blocking Operations

The main obstacle to achieving self-stabilization is that the MVC algorithm blocks on many operations. As mentioned, there are operations where the node waits for a result or a reply from another node. Specifically, the MVC *propose()* as well as *vbbBroadcast()* block until there is a result. Besides these operations, BRB and VBB are designed to block until *brbDeliver(),* and *vbbDeliver()* provide a result. We will see in the next chapter how these operations are transformed into non-blocking.

3.3.1.2 Blocking Reliable channels and transient errors

The reliable channels are another algorithm component that can block the system during execution. In order to assure reliability, the channels must either block until there is a reply or resend a message in case the first one was omitted. The solution here is to gather all messages into one big message and resend it in a repeated for-loop; thus, we can assume that messages will be eventually transferred. This solution also solves the problem of transient errors, where a transient error in our case can cause the program to come into an error state. For example, a state variable change that does not let the algorithm finish or corruption on the program counter can block the system on a communication procedure. To avoid this, all the code parts that a node would wait for are converted to if-statements, and in combination with the for-loop, the system will never block.

3.3.2 The Self-stabilizing Byzantine-tolerant Multivalued Consensus Algorithm

We can now show the proposing Self-stabilizing Byzantine-tolerant Multivalued Consensus Algorithm by Duvignau, Schiller, and Raynal. Each algorithm's conversion and building block is analyzed in the following sub-sections.

3.3.2.1 General Structure, variables, and types

In contrast to how the algorithms work, as shown in Section 3.2, the Self-stabilizing Byzantine-tolerant Multivalued Consensus works in an iterating way.

First, to avoid blocking during many broadcasting operations in the algorithms, all messages are unified in one big message and then sent. For that, a structure *msg*[][][] is used for holding all messages that are sent and received. This 3-dimensional structure stores the messages, according to the following logic: *msg*[*nodeId*][*vbbType*][*brbType*]. First, messages are grouped by nodes. Assuming a node with id *i*, *msg*[*i*] contains all the messages that node *i* is supposed to send. For every other node with id *j* ≠ *i*, the received messages are kept in *msg*[*j*]. Next, the VBB-broadcast messages and finally BRB-broadcast messages are stored; for example, the received *VBB_INIT* messages of node *j* are stored in *msg*[*j*][*VBB_INIT*], and the *BRB_READY* messages of *VBB_INIT* messages are stored in *msg*[*j*][*VBB_INIT*][*BRB_READY*]. As already said, there are two (2) types of VBB messages and three (3) of BRB; *VBB_INIT*, *VBB_READY* for VBB-broadcast, and *BRB_INIT*, *BRB_ECHO*, *BRB_READY* for BRB-broadcast.

There is a background task that is responsible for handling received messages and storing them in the *msg*[][][] structure. In each iteration, the algorithm operations take place by checking the received messages in the *msg*[][][], and all new messages that should be sent are added to that structure. At the end of each iteration, the node's messages stored in *msg*[*i*] are broadcasted. This operation happens in a forever for-loop.

The algorithm also uses a Byzantine-tolerant Binary Consensus algorithm, which is formatted as a binary consensus object **bcO**, whose initial state is inactive (⊥) and becomes active when *bcO.propose()* is invoked. The binary consensus algorithm is executed as a background task when activated, hence not blocking the main MVC algorithm. The *bcO.result()* operation returns the decided binary value.

Mentioned variables and types are shown below.

```
types:
    brbTypes = {BRB_INIT, BRB_ECHO, BRB_READY}
    vbbTypes = {VBB_INIT, VBB_VALID}
variables:
    msg[][][] // most recently sent and received messages
    bcO // binary object, ⊥ when not running
background task:
    when message m arrived from node with nodeId:
        msg[nodeId] = m
```

Figure 11: Self-stabilizing MVC Variables and types

3.3.2.2 Self-stabilizing BRB-broadcast and VBB-broadcast

The self-stabilizing BRB-broadcast and VBB-broadcast algorithms are shown below. We see how their broadcast and deliver operations do not directly broadcast or wait for messages but read/write to the *msg*[][][] structure.

```
// BRB-broadcast
brbBroadcast(vbbType, v):
    msg[i][vbbType][BRB_INIT] = v

brbDeliver(vbbType, nodeId):
    if (nodeId, m) in msg[*][vbbType][BRB_READY]
      of 2t+1 different nodes:
        return m
    else:
        return ⊥

// VBB-broadcast
vbbEq(vbbType, v):
    if brbDeliver(vbbType, nodeId) == v
      by at least n-2t nodes:
        return true
    else:
        return false

vbbDiff(vbbType, v):
    if brbDeliver(vbbType, nodeId) ≠ v
      by at least t+1 nodes:
        return true
    else:
        return false

vbbEcho(vbbType):
    if brbDeliver(vbbType, nodeId) ≠ ⊥
      by at least n-t nodes:
        return true
    else:
        return false

vbbBroadcast(v):
    brbBroadcast(VBB_INIT, (i, v))

vbbDeliver(nodeId):
    if msg[nodeId][VBB_INIT][BRB_INIT] == ⊥ &&
      msg[nodeId][VBB_VALID][BRB_INIT] ≠ ⊥:
        return Ψ

    initValue = brbDeliver(VBB_INIT, nodeId)
    validValue = brbDeliver(VBB_VALID, nodeId)

    if ¬(initValue ≠ ⊥ && validValue ≠ ⊥):
        return ⊥
    if validValue == true && vbbEq(VBB_INIT, initValue):
        return initValue
    if validValue == false && vbbDiff(VBB_INIT, initValue):
        return Ψ
    if vbbEcho(VBB_VALID):
        return Ψ
    return ⊥
```

Figure 12: Self-stabilizing BRB-broadcast and VBB-broadcast algorithms

3.3.2.3 Consistency tests

For the algorithm to be self-stabilizing, consistency tests must be in place to detect and recover from arbitrary transient errors. The consistency tests are shown in Figure 13. The first two consistency tests (*brbMessagesConsistencyTest()* and *vbbValidMessagesConsistencyTest()*) are used for BRB-broadcast messages and VBB-broadcast messages, respectively, and check for inconsistency in *BRB_ECHO*, *BRB_READY* and *VBB_VALID* messages that a node is about to send. The last consistency test checks for inconsistencies in received messages. In case inconsistency is detected, all the equivalent messages are erased. This message purge is safe since the algorithm is designed to resend all messages again.

```
// Consistency tests
brbMessagesConsistencyTest(vbbType):
    for message in msg[i][vbbType][BRB_ECHO]:
        if not equivalent BRB_INIT message:
            clear msg[i]
    for message in msg[i][vbbType][BRB_READY]:
        if ¬(at least (n+t)/2 equivalent BRB_ECHO messages ||
          at least t+1 equivalent BRB_READY messages):
            clear msg[i]

vbbValidMessagesConsistencyTest(vbbType):
    initValue = msg[i][vbbType][BRB_INIT]
    validValue = msg[i][VBB_VALID][BRB_INIT]
    if initValue == ⊥:
        clear msg[i]
    if validValue ≠⊥:
        if ¬(vbbEcho(vbbType) &&
          validValue == vbbEq(VBB_INIT, initValue)):
            clear msg[i]

receivedMessagesConsistencyTest(vbbType):
    for every nodeId in msg[][][]:
        initValue = msg[nodeId][VBB_INIT][BRB_INIT]
        if initValue == ⊥ || initValue sender ≠ nodeId:
            clear msg[nodeId]

        for message in msg[nodeId][vbbType][BRB_ECHO]:
            if message sender duplicate:
                clear msg[nodeId]

        for message in msg[nodeId][vbbType][BRB_READY]:
            if message sender duplicate:
                clear msg[nodeId]
```

Figure 13: Consistency test for self-stabilizing MVC

3.3.2.4 Self-stabilizing Multivalued Consensus

On top of the self-stabilizing Byzantine and intrusion-tolerant protocol stack, we have the self-stabilizing Multivalued consensus. There is a minor change in the Binary Consensus algorithm for the self-stabilizing MVC to work. The *binPropose()* shown in Figure 9 changes and is executed asynchronously instead of blocking until a binary value is decided. When a value is decided, it is stored in a variable and can be retrieved asynchronously through *bcO.binResult()*.

In Figure 14, we see the logic of the self-stabilizing MVC algorithm. Specifically, we see the **propose()** operation in lines 111-112, which invokes the vbbBroadcast() operation. In lines 114-128, we have the **result()** operation, which is meant to be invoked asynchronously to the MVC() operation. If there is a decided value, the value is returned, otherwise ⊥ or Ψ, depending on the BC result.

The main algorithm logic **MVC()** is in lines 130-156, executed in a never-ending for-loop. At the start of each iteration, in lines 133-135, the consistency tests are executed. In lines 137-151, we see all the relevant checks performed on received BRB and VBB messages. In lines 153-154, we see the check on VBB messages and the proposal to the Binary Consensus object, which is the result of the **bp()** test found in lines 103-109. Finally, the node broadcasts all of its messages at the end of each iteration, in line 156.

```
// MVC
bp():
    if only one distinct non-Ψ value vbbDelivered &&
      vbbDelivered by at least n-2t nodes:
        return true
    else:
        return false

propose(v):
    vbbBroadcast(v)

result():
    if bcO == ⊥ || bcO.binResult() == ⊥ ||
      msg[i][VBB_INIT][BRB_INIT] == ⊥:
        return ⊥

    if bcO.result() == Ψ:
        return Ψ

    if more than n-t values ∉ {⊥, Ψ} vbbDelivered && ¬bp():
        return Ψ

    if there is v vbbDelivered by at least n-2t nodes:
        return v

    return ⊥

MVC():
    do forever:
        for each vbbType in vbbTypes:
            brbMessagesConsistencyTest(vbbType)
            vbbValidMessagesConsistencyTest(vbbType)
            receivedMessagesConsistencyTest(vbbType)

            if received BRB_INIT message:
                add equivalent BRB_ECHO message to
                msg[i][vbbType][BRB_ECHO]

            if same message msg[*][vbbType][BRB_ECHO]
              by at least (n+t)/2 nodes ||
            if same message msg[*][vbbType][BRB_READY]
              by at least t+1 nodes:
                add equivalent BRB_ECHO message to
                msg[i][vbbType][BRB_READY]

        if vbbEcho(messages.VBB_INIT) == true:
            validValue = vbbEq(messages.VBB_INIT,
              msg[i][VBB_INIT][BRB_INIT])
            brbBroadcast(messages.VBB_VALID, validValue)

        if n-t values ∉ {⊥, Ψ} vbbDelivered:
            bcO.binPropose(bp())

        broadcast msg[i]
```

Figure 14: Self-stabilizing MVC algorithm

Chapter 4

Implementation Details

4.1 Technical Details

As mentioned in Chapter 2, for the implementation, we used the Go Programming Language for its simplicity and strong concurrency tools, like channels and goroutines, which are part of the language syntax. It also helped us be more precise with our comparison with other implementations. Specifically, we used the *go1.18* version.

We used the ZeroMQ message-passing library for its high performance on asynchronous communication operations for the communication between the processes. The ZeroMQ is provided in the most famous languages and almost every operating system. We used the equivalent ZeroMQ library in Go, which can be found on Github under pebbe/zmq4 v1.2.8 [36].

The algorithm was implemented and tested on Visual Studio Code [37] on Linux Ubuntu 20.04 64-bit operating system [38].

4.2 Project Overview

To implement the project's algorithm, we created a module named **self-stabilizing-mvc**. The module contains five (5) packages, including the *main* package, where the source file with the main function is placed, alongside four (4) other packages necessary for executing the algorithm. The *utils* package contains essential utility functions, i.e., for serializing and deserializing and parsers for the program cmd arguments and config files. Everything related to ZeroMQ sockets and communication among processes is placed in the *network* package. Next, the *messages* package contains structure implementations and logic related to the BC and MVC messages and message passing between internal goroutines. Lastly, every MVC and BC protocol stack protocol is implemented in the consensus package. Every package, alongside each source file, contains another unit and automation test file, following the naming convention source-file-name**_test**.go, i.e., the *network* package contains a source file named network-utils.go, and the unit test file network-utils_test.go. The project can be found on Github [39].

4.3 Implementation challenges

While studying the algorithm [6], an initial challenge was the existence of some ambiguities and typographical errors in the pseudocode and explanations. These mismatches led to bugs during implementation, but we could resolve them through additional discussions and studying of the relevant algorithms.

A primary challenge we faced was implementing inter-node communication since it should be asynchronous, fast, and not over-engineered. We tackled this issue quickly by using the already described ZeroMQ messaging library, which covered our needs for this implementation requirement. Thanks to ZeroMQ, the provided sockets, and easy-to-use functions, we eventually implemented the network architecture described in the next section, which covered all of our needs.

Another challenge faced when a concurrent algorithm is implemented is that it was not straightforward how to debug our implementation in cases of errors. For that, we took advantage of the algorithms' iterative design, and in each iteration, we could print the received and broadcasted messages to detect issues.

There was an added risk because every added functionality was not tested in a real-life-like environment and only on the local workstation. This could lead to missing bugs that would only be visible during experimental analysis. For that, we tried running multiple scenarios on at least two conventional laptops with up to 75 nodes. Eventually, there were not any issues missed.

4.4 Communication structure

For the communication between the nodes, ZeroMQ sockets were explicitly used. Since the studied architecture assumes an asynchronous message-passing system, we were flexible with the socket implementation. For this, every node needs two types of sockets, one for receiving messages from other nodes and another set of nodes that can be used for sending messages asynchronously.

As we can observe in the resulting algorithm in Figure 14, we only see how messages are sent, and there is no code for receiving messages needed for the algorithm. The reason is that receiving messages is considered a background task, running on another thread that cannot block the algorithm's execution. For simplicity, the receiving operation can even be a blocking operation. When a message is received, it is added to the received messages collection.

On the other side, sending messages is part of the algorithm, meaning that a node must be able to send a message to another node and not block until the other node receives the message and replies. The property of asynchronously sending messages combined with the repeating transmitting absolves the node from caring if the message is received from the other side. Hence the only action needed by the node is to "publish" its messages in every iteration.

Considering the above, ideally, the ZeroMQ PUB and SUB sockets should be used for the algorithm, but for simplicity, we used the classic REQ and REP sockets with the *zmq4.DONTWAIT* flag of the implemented library [36], creating the same behavior. Specifically, each node has a REP socket listening to incoming messages on a different thread, and when a message is received, it is added to the received messages collection. Moreover, each node has a set of *n-1* REQ sockets for communicating with other nodes' REP sockets. Messages are sent through each REQ socket, and since the REQ sockets need to perform a *receive* operation between its *sents*, a dummy receive with the *DONTWAIT* flag is performed. We can see the complete communication structure in Figure 15.

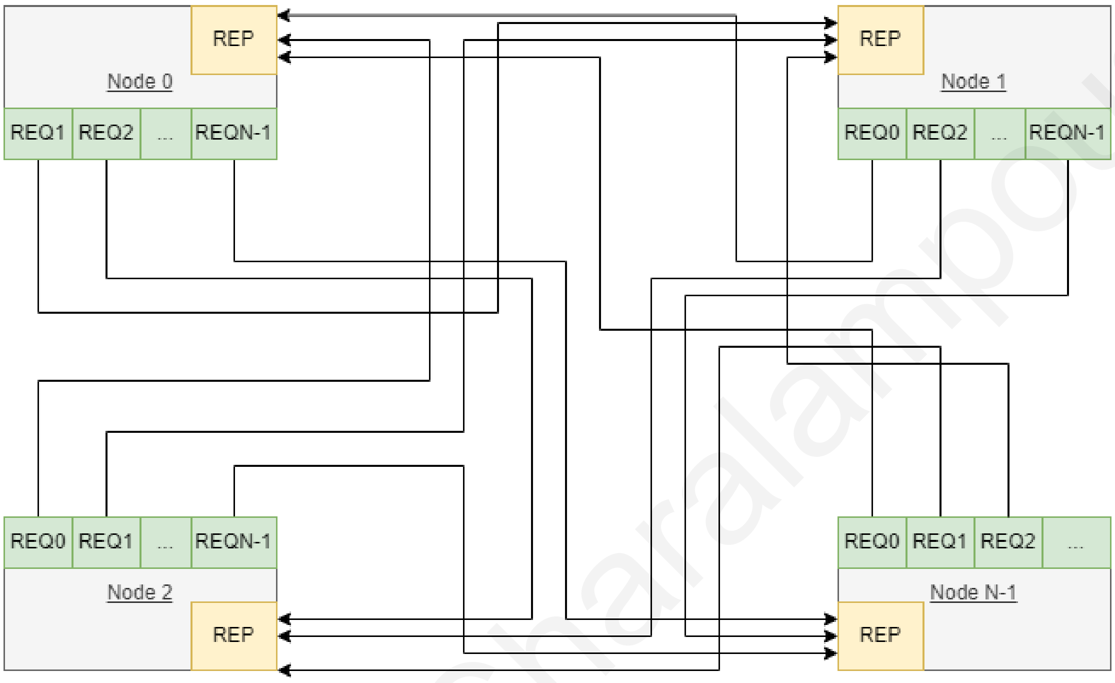

Figure 15: The implemented network architecture

4.5 Protocols Implementation

We can now see the implementation of all the protocols using the Go programming language, starting from the bottom of the protocols stack. We will see code parts for each algorithm and how they match the resulting algorithm. There are enough comments on each code shown for those unfamiliar with the Go syntax.

4.5.1 Messages

Starting with the messages, we have two (2) structs used as collections to hold all the exchanged MVC and BC messages. **BCMessagesRegistry** and **MvcMessagesRegistry,** as shown in Figure 16 and Figure 17, respectively, are used for holding the messages. Whenever a node is said to send a message, that message is added to the equivalent registry. At the end of each for-loop iteration, the current node's content in the registry is sent to all nodes. Because there are two types of messages and, for simplicity, only one REP socket per node, messages are wrapped in another struct containing their message type, *MSG_TYPE_MVC* or *MSG_TYPE_BC,* so the receiving node knows how to deal with each message.

```go
// Binary consensus message types
var BcTypes = []int{BC_EST, BC_AUX}
const (
    BC_EST = iota
    BC_AUX
)

// Struct that holds all messages sent and received
var BcMessagesRegistry struct {
    // maps nodeId to its messages
    NodesMessages map[string]*BcRoundMessages
    BinValues     map[int][]bool // the bin_values collection
    CurrentNodeId string // the current nodeId
    // mutex for accessing and modifying the structure
    Mutex         *sync.Mutex
}

// struct that maps the BC messages of a node to the round
// that were sent/received
type BcRoundMessages struct {
    RoundMessages map[int]*BcMessages
}

// all messages of a specific node that could be
// sent/received during a round
type BcMessages struct {
    EstMessages *[]bool
    AuxMessage *[1]bool
}
```

Figure 16: Implementation of the BC Messages struct

```go
// VBB Message types
var (
    BrbTypes = []int{BRB_INIT, BRB_ECHO, BRB_READY}
    VbbTypes = []int{VBB_INIT, VBB_VALID}
)
const (
    BRB_INIT = iota
    BRB_ECHO
    BRB_READY
)
const (
    VBB_INIT = iota
    VBB_VALID
)

// Holds all MVC shared messages (msg[][][])
var MvcMessagesRegistry struct {
    // maps nodeId to its messages
    NodesMessages map[string]VbbMessages
    CurrentNodeId string // the current nodeId
    // mutex for accessing and modifying the structure
    Mutex         *sync.RWMutex
}

// Holds all VBB Messages that were BRB Broadcasted
type VbbMessages struct {
    // 0: VBB Init Message, 2: VBB Valid Message
    BrbMessages [2]BrbMessages
}

// Holds all BRB Messages that were broadcasted
type BrbMessages struct {
    // 0: BRB Init Message,1: BRB Echo Messages,2: BRB Ready Messages
    Messages [3]*[]Message
}

//message containing sender nodeId and the value that it broadcasted
type Message struct {
    NodeId string
    Value  string
}
```

Figure 17: Implementation of the MVC Messages struct

4.5.2 Asynchronous message-passing system

Every message object transferred is serialized and deserialized using the *encoding/gob* library [40], so the equivalent *SendBytes()* and *RecvBytes()* of ZeroMQ sockets are used. There is a goroutine running in the background in which the REP socket listens to incoming messages and forwards them to handlers. Handlers extract the sender, and messages are added to the equivalent registry. When broadcasting messages, a non-blocking send is called for each REQ socket. Below we see the REP and REQ sockets' functionality.

```go
// The given REP keeps listening to incoming messages and after
// checking its type, it forwards to the equivalent handler
func ReplySocketReader(replySocket *zmq4.Socket) {

    for {

        // blocking - reading for incoming messages
        msgBytes, _ := replySocket.RecvBytes(0)

        // deserialization of message
        message, err := utils.ToObject(msgBytes)
        if err != nil {
            logger.WarnLogger.Println("Decoding failed", err)
            continue
        }

        // message is forwarded to equivalent message handler
        if message.MsgType == messages.MSG_TYPE_MVC {
            messages.MVCChannel <- message
        } else if message.MsgType == messages.MSG_TYPE_BC {
            messages.BCChannel <- message
        }

        // dummy non blocking send, so that socket state resets
        replySocket.Send("", zmq4.DONTWAIT)
    }
}
```

Figure 18: Implementation of the REP socket

```go
// Broadcasts the payload to all given sockets
func BroadcastToAll(remoteSockets []*zmq4.Socket, msg interface{}) {

    // serialization of message
    payload, err := utils.ToBytes(msg)
    if err != nil {
        logger.WarnLogger.Println("Encoding failed", err)
        return
    }

    // send to every REQ socket
    for _, remoteSocket := range remoteSockets {

        // mutex used in case BC broadcasts
        // at the same time with MVC protocol
        remoteSocketsMutex.Lock()

        // non-blocking send
        remoteSocket.SendBytes(payload, zmq4.DONTWAIT)

        // dummy non-blocking receive, to restart socket state
        remoteSocket.Recv(zmq4.DONTWAIT)

        remoteSocketsMutex.Unlock()
    }
}
```

Figure 19: Implementation of broadcast functionality using REQ sockets

4.5.3 Byzantine Reliable Broadcast

The implementation of the BRB operations is shown below. During the *brbBroadcast* operation, the equivalent message is added to the **MvcMessagesRegistry**. During *brbDeliver*, the messages in the registry are checked, and the corresponding value is returned.

```go
// brbBroadcasts the given message
func BrbBroadcast(vbbType int, message Message) {

    validateVbbType(vbbType)

    currentNodeId := ConsensusContext.NodeId

    nodeMessages := MvcMessagesRegistry.NodesMessages[currentNodeId]
    // the brbMessages of the equivalent VBB messages are fetched
    brbInitMessages := BrbMessages[vbbType].Messages[BRB_INIT]

    // new brb
    if len(*brbInitMessages) == 0 {
        *brbInitMessages = append(*brbInitMessages, message)
    } else {
        (*brbInitMessages)[0] = message
    }
}

// Checks if vbbType message of nodeId given is brbDelivered.
// The brbDeliver value is returned else NonDecidedValue "{}"
func BrbDeliver(vbbType int, nodeId string) string {

    validateVbbType(vbbType)

    // Check if message exists in node given
    nodesMessages := MvcMessagesRegistry.NodesMessages
    nodeMessages, exists := nodesMessages[nodeId]
    if !exists {
        return NonDecidedValue
    }

    brbInitMessages := nodeMessages.BrbMessages[vbbType].
        Messages[BRB_INIT]
    if brbInitMessages == nil || len(*brbInitMessages) == 0 {
        return NonDecidedValue
    }

    expectedMessage := (*brbInitMessages)[0]

    currentNodeId := ConsensusContext.NodeId
    minimumReadies := 2*ConsensusContext.MaximumByzantines + 1

    // finds number of readies sent by other nodes, matching the
    // node's brbInit value
    actualReadies := 0
    for checkedNode, checkedNodeMessages := range nodesMessages {

        // ignore current node and the node whom value is checked
        // for delivery
        if checkedNode == currentNodeId || checkedNode == nodeId {
            continue
        }

        brbReadyMessages := checkedNodeMessages.BrbMessages[vbbType].
```

```
            Messages[messages.BRB_READY]
        // if the checked node has sent a matching READY message,
        // the counter of READY sent is increased
        if brbReadyMessages != nil && utils.SliceContains(
            *brbReadyMessages,
            expectedMessage) {

                actualReadies++
        }
    }

    // if enough READY received, the value is brbDelivered
    if actualReadies >= minimumReadies {
        return expectedMessage.Value
    } else {
        return NonDecidedValue
    }
}
```

Figure 20: The implemented BRB-broadcast algorithm

4.5.4 Validated Byzantine Broadcast

Next, we see the implementation of VBB operations, which depends on the BRB operations described in the previous section.

```go
// vbbBroadcasts the given value
func VbbBroadcast(value string) {

    currentNodeId := ConsensusContext.NodeId

    message := Message{
        NodeId: currentNodeId,
        Value:  value,
    }

    BrbBroadcast(VBB_INIT, message)
}
// Checks if the message of given node is vbbDelivered.
// If validated, then the message sent from the node is returned.
// If not yet delivered, then the NonDecidedValue "{}" is returned.
// If the message is invalid, then ErrorValuePsi "#PSI" is returned.
func VbbDeliver(nodeId string) string {

    nodeMessages := MvcMessagesRegistry.NodesMessages[nodeId]
    vbbInitMessages := nodeMessages.BrbMessages[VBB_INIT].
        Messages[BRB_INIT]

    if vbbInitMessages == nil {
        return ErrorValuePsi
    }

    // checks if both VBB_INIT and VBB_VALID are brbDelivered
    initDeliveredValue := BrbDeliver(VBB_INIT, nodeId)
    validDeliveredValue := BrbDeliver(VBB_VALID, nodeId)

    if initDeliveredValue == NonDecidedValue ||
        validDeliveredValue == NonDecidedValue {
            return NonDecidedValue
    }

    if validDeliveredValue == "true" &&
        vbbEq(VBB_INIT, initDeliveredValue) {
            return initDeliveredValue
    }

    if validDeliveredValue == "false" &&
        vbbDiff(VBB_INIT, initDeliveredValue) {
            return ErrorValuePsi
    }

    if vbbEcho(VBB_VALID) {
        return ErrorValuePsi
    }

    return NonDecidedValue
}
```

Figure 21: The implemented VBB-broadcast algorithm

4.5.5 Randomized Binary Consensus

Our implementation of Binary Consensus is a transformation of the existing solution provided by Petrou [7] to fit our asynchronous message-passing. Flow is similar to MVC. Each round runs in a for-loop until progressing to the next round or deciding on a value. The **BcMessagesRegistry** is filled during Binary-value broadcast, and in every iteration, the messages are broadcasted to all nodes. This module is not self-stabilizing but does not affect our MVC implementation, as they are executed in parallel and independently. BVB and BC implementations are shown in Figure 8 and Figure 9.

```go
// bvbBroadcasts the value of given type for the round provided
func BvbBroadcast(round int, value bool, bcType int) {

    broadcastBinaryValue(round, value, bcType)

    checkReceivedEstValues(round)
}

// Check received EST values for rebroadcasting or
// adding to bin_values
func checkReceivedEstValues(round int) {

    valuesCounter := GetBcValuesCounterForMsgType(round, BC_EST)
    maximumByzantines := ConsensusContext.MaximumByzantines

    for value, occurrences := range valuesCounter {

        if occurrences >= maximumByzantines+1 {
            broadcastBinaryValue(round, value, BC_EST)
        }

        if occurrences >= 2*maximumByzantines+1 &&
            !utils.SliceContains(
                BcMessagesRegistry.BinValues[round],
                value) {

            BcMessagesRegistry.BinValues[round] = append(
                BcMessagesRegistry.BinValues[round],
                value)
        }
    }
}
```

Figure 22: The implemented BVB-broadcast algorithm

```go
//implementation of the binPropose operation of Binary Consensus
func BcPropose(value bool) {
    // initial estimation
    estimation := value
    for round := 1; ; round++ {
        for {
            // BC PHASE 1
            BcMessagesRegistry.Mutex.Lock()
            BvbBroadcast(round, estimation, BC_EST)
            // sends all node messages to other nodes
            broadcastBcMessages()

            // can't continue to second phase, till there are
            // bin_values
            if len(BcMessagesRegistry.BinValues[round]) <= 0 {
                BcMessagesRegistry.Mutex.Unlock()
                continue
            }

            // BC PHASE 2
            auxValue := BcMessagesRegistry.BinValues[round][0]
            broadcastBinaryValue(round, auxValue, BC_AUX)
            auxValues, totalAuxValues := getAuxValues(round)

            BcMessagesRegistry.Mutex.Unlock()

            totalNodes := ConsensusContext.NumberOfNodes
            byzantines := ConsensusContext.MaximumByzantines

            if totalAuxValues < totalNodes-byzantines {
                continue
            }
            // BC PHASE 3
            coinValue := flipCommonCoin(round)
            if len(auxValues) == 2 {
                estimation = coinValue
            } else if auxValues[0] != coinValue {
                estimation = auxValues[0]
            } else {
                decide(auxValues[0])
            }
            break
        }
    }
}

// Returns the decided value or NonDecidedValue "{}"
func BcResult() string {
    bcMutex.Lock()
    defer bcMutex.Unlock()
    result := decidedValue
    return result
}
```

Figure 23: The implemented BC algorithm

4.5.6 Multivalued Byzantine Consensus

Finally, Multi-valued Byzantine Consensus is built above all of the previous protocols. Below we can see the *propose()* and *result()* operations of MVC and the main core of the whole algorithm.

```go
// Implementation of propose() operation
func MvcPropose(value string) {

    VbbBroadcast(value)
}

// Implemenation of Self-stabilizing, Byzantine-no-intrusion-tolerant
// Multivalued Consensus
func MultiValueConsensus(value string) {

    for {

        MvcMessagesRegistry.Mutex.Lock()

        // perform consistency tests and check for forwarding
        // messages for each VBB message type
        for _, vbbType := range VbbTypes {

            consistencyTestBrbMessages(vbbType)
            consistencyTestVbbValidMessages(vbbType)
            consistencyTestReceivedMessages(vbbType)

            // checks if there are new BRB-INIT messages, in order
            // to send equivalent echo messages
            checkForSendingEcho(vbbType)
            // checks if there are enough BRB-ECHO messages, in order
            // to send equivalent READY messages
            checkForSendingReady(vbbType)
        }

        // re-proposing in case proposed value erased during
        // consistency tests
        MvcPropose(value)

        // checks if there are enough VBB-INIT that were brbDelivered,
        // in order to send first VBB-VALID message
        checkForBroadcastingVbbValid()

        // checks if there are enough VBB-delivered messages, in order
        // to propose to BC object
        checkForBcProposing()

        // broadcast all MVC messages of current node
        broadcastCurrentNodeMvcMessages()
        MvcMessagesRegistry.Mutex.Unlock()
    }
}
```

```go

// Get the MVC decided value.
// If no value decided yet, NonDecided "{}" is returned.
// If consensus could not be reached, ErrorValue Psi "#PSI"
// is returned.
func MvcResult() string {

    MvcMessagesRegistry.Mutex.RLock()
    defer MvcMessagesRegistry.Mutex.RUnlock()

    currentNodeId := ConsensusContext.NodeId
    currentNodeMessages := MvcMessagesRegistry.
                               NodesMessages[currentNodeId]
    initMessage := currentNodeMessages.BrbMessages[VBB_INIT].
                       Messages[BRB_INIT]

    if !isBcObjectActive()
        || BcResult() == NonDecidedValue
        || initMessage == nil {
            return NonDecidedValue
    }

    if BcResult() != "true" {
        return ErrorValuePsi
    }

    numberOfNodes := ConsensusContext.NumberOfNodes
    maximumByzantines := ConsensusContext.MaximumByzantines

    vbbDeliveredValues, vbbDeliveredCounter := getVbbDeliveredValues()
    if vbbDeliveredCounter >= numberOfNodes-maximumByzantines &&
        !bp() {
            return ErrorValuePsi
    }

    minimumValueOccurrences := numberOfNodes - 2*maximumByzantines
    for value, occurrences := range vbbDeliveredValues {

        if occurrences >= minimumValueOccurrences {
            return value
        }
    }

    return NonDecidedValue
}
```

Figure 24: The implemented MVC algorithm

4.6 Execution & Configuration Details

The project can be built with the command:

```
$ go build
```

An executable named *self-stabilizing-mvc* is created, which starts a node when executed. Two main execution parameters are needed for the node to start; a **port number** to where the REP socket will bind and a path to a **configuration file** containing the hostname and port of each other node to be reached with the REQ sockets. There are another two execution arguments, one for starting a Byzantine node following a specific scenario and one for simulating a transient error inside a node. The parameters and configuration details are analyzed next.

4.6.1 Port and Remote Nodes configuration

A configuration file is used for each node to know its remote nodes. The file contains in each line a remote node, in the syntax of *<protocol>://<host>:<port>*. Figure 25 shows an example of a config file of nodes that connect to five (5) remote hosts.

```
# configuration.conf file

http://node1:1234
http://node2:1234
http://node3:1234
http://node4:1234
http://node5:1234
```

Figure 25: Example of a network configuration file

The following needed property is the node's port to listen for receiving messages from remote nodes. To start a node, the following command is executed:

```
$ ./self-stabilizing-mvc –p 8080 –f configuration.conf
```

The above will start a node listening for incoming messages to port 8080 and send messages to remote nodes specified in the *configuration.conf* file.

4.6.2 Byzantine nodes and Transient Errors

The last two execution arguments can be used to start byzantine nodes or simulate transient errors. A transient error can be simulated by giving the flag *–t,* and Byzantine nodes can apply one of the attack scenarios analyzed in Chapter 6 by giving the flag *–b* and specifying one of the following values during execution:

Scenario	Flag value
Failure-free Scenario	0
Byzantines Idle Attack Scenario	1
Byzantines Half & Half Attack Scenario	2
Byzantines Random Messages Attack Scenario	3

The following command will start a byzantine node receiving messages on port 8080, and its remote nodes are found in file configuration.conf. The node will apply the Byzantines Half & Half Attack scenario.

```
$ ./self-stabilizing-mvc –p 8080 –f configuration.conf –b 2
```

On the other hand, the following command will start a non-Byzantine node receiving messages on port 8080, and its remote nodes are found in file configuration.conf. Also, a transient internal error will happen during its execution.

```
$ ./self-stabilizing-mvc –p 8080 –f configuration.conf –t
```

Chapter 5

Self-stabilization and Byzantine Fault Tolerance Scenarios and Validation

5.1 Introduction

In this chapter, we will look at different use cases handled by the algorithm to reach consensus eventually. We tested every case during implementation and provided automation tests that simulated the bare minimum of each case to assure validity and easy repetition of scenarios. Therefore, for each scenario presented, we show the equivalent automation test, in which one node performs MVC, and we simulate the rest of the nodes by inserting messages in the **MvcMessagesRegistry**. In Section 5.2, we see some scenarios in the presence of byzantine attacks, and in Section 5.3, we look at consensus scenarios in different cases where transient errors happen. Finally, in Section 5.4, we see scenarios where consensus is reached even when transient errors and byzantine faults happen simultaneously.

5.2 Byzantine Fault Tolerance Scenarios

Byzantine Fault Tolerance Scenarios contain scenarios where Byzantine nodes act maliciously, sabotaging during MVC in order the system does not come to consensus. Our tests consider that Byzantine nodes interfere only with the messages they send, meaning that they can either send invalid messages, omit them, or even not respond.

5.2.1 Byzantine node that is Idle or proposes an invalid value

In the first case, we test that consensus is reached if there are Byzantine nodes as long as the number of Byzantine nodes is $t < n/3$. Here, we have six nodes, one being

Byzantine, and since *1 < 6/3(=2)*, consensus is reached without a problem. Every correct node proposes the value *42*.

```
func TestMultiValueConsensusWithExpectedFaultyWhichIsIdle (t
*testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 6, NO_ATTACK)
    proposedValue := "42"

    sendMvcMessagesOf4CorrectNodes()

    go MultiValueConsensus(proposedValue)

    sendBcMessagesOf4CorrectNodes()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

Similarly, consensus is still reached if the Byzantine node sends a random invalid value. In this case, the Byzantine node proposes the value *7*.

```go
func TestMultiValueConsensusWithExpectedFaultyProposingInvalidValue(t
*testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 6, NO_ATTACK)
    proposedValue := "42"

    sendMvcMessagesOf4CorrectAndOneFaultyNode()

    go MultiValueConsensus(proposedValue)

    sendBcMessagesOf4CorrectAndOneFaultyNode()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.2.2 Byzantine node sending invalid ECHO

The next test is similar to the previous, but instead of having a Byzantine node that proposes an invalid value, now the Byzantine node sends invalid ECHO messages. Specifically, the Byzantine node with ID 6 will send an ECHO message saying that the node with ID 2 proposed the value *7*. Again consensus is reached on value *42*.

```go
func TestMultiValueConsensusWithExpectedFaultyWhichSendsInvalidEcho(t
*testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 6, NO_ATTACK)
    proposedValue := "42"

    sendMvcMessagesOf4CorrectAndOneFaultyNode()

    // invalid echo message, node 6 says that node 2 sent BRB_INIT
    // with value 7
    addMvcMessage("6", VBB_INIT, BRB_ECHO, "2", "7")

    go MultiValueConsensus(proposedValue)

    sendBcMessagesOf4CorrectAndOneFaultyNode()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.2.3 No Consensus with more Byzantines than expected

In this last test, we verify that consensus cannot be reached if the number of Byzantine nodes exceeds the third of all nodes. We specify an environment of seven nodes with four idles that do not send or reply to any messages, where *4 > 7/3*. Therefore, consensus is never reached.

```go
func TestMultiValueConsensusWithMoreFaultyThanExpected(t *testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 7, NO_ATTACK)
    proposedValue := "42"

    // only 2 + 1 correct, 4 missing considered faulty
    sendMvcMessagesOf2CorrectNodes()

    go MultiValueConsensus(proposedValue)

    // for 5 seconds, every 100ms we get the result and we verify
    // that every time consensus is not reached
    for i := 1; i < 50; i++ {

        time.Sleep(100 * time.Millisecond)
        assert.Equal(t, NonDecidedValue, MvcResult())
    }
}
```

5.3 Self-stabilization on transient errors

Of course, one main property we tested and assured is self-stabilization in case of transient errors. In these automation tests, we inject transient errors by modifying the messages that a node holds. These errors should not be confused with the Byzantine attacks, meaning that alternations in messages happen internally and can even alternate messages sent by a correct node. Below we present three examples of such errors and how the system self-stabilizes after each one. The cases are many more, but they are identified and handled similarly.

5.3.1 Transient error on the proposed value

As part of this scenario, after the node made its value proposal to MVC, we overwrite that proposal with an invalid value. We assume three nodes in this setup, and by injecting the error, consensus should not be reached without self-stabilization since one node will "act" as faulty. This error is handled by proposing the correct value for every iteration in the for-loop. In the end, we see that consensus is reached.

```go
func TestMultiValueConsensusWithTransientErrorOnInit(t *testing.T) {

    setupTestingConsensusContext(t, 3, NO_ATTACK)
    proposedValue := "42"
    currentNodeId := ConsensusContext.NodeId

    sendMvcMessagesOf2CorrectNodes()

    go MultiValueConsensus(proposedValue)

    // transient error - overwrite init message
    addMvcMessage(currentNodeId,VBB_INIT,
        BRB_INIT, currentNodeId, "1231234")

    sendBcMessagesOf2CorrectNodes()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.3.2 Transient Error on ECHO message

Here we inject an invalid echo message, stating that nodes 2 and 3 proposed a different value. Because of the consistency tests, the ECHO messages are not matching the INIT messages. Therefore by rule, all the current node's messages are deleted, starting the mvcPropose from the beginning. Similarly to the previous test, we have three nodes in total, and the current node is supposed to act faulty. We can verify again that consensus has been reached.

```go
func TestMultiValueConsensusWithTransientErrorOnEcho(t *testing.T) {

    setupTestingConsensusContext(t, 3, NO_ATTACK)
    proposedValue := "42"
    currentNodeId := ConsensusContext.NodeId

    sendMvcMessagesOf2CorrectNodes()

    // transient error - inject invalid echo messages
    addMvcMessage(currentNodeId, VBB_INIT,BRB_ECHO, "2", "1231234")
    addMvcMessage(currentNodeId, VBB_INIT,BRB_ECHO, "3", "1231234")

    go MultiValueConsensus(proposedValue)

    sendBcMessagesOf2CorrectNodes()

    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.3.3 Transient error on validity check

We showed that during Validated Byzantine Broadcast, when enough values are brbDelivered, a validity test takes place, and the result is proposed to the Binary Consensus object. This test shows how the system recovers if the validity check results are corrupted.

```go
func TestMultiValueConsensusTransientErrorOnValidInit(t *testing.T) {

    setupTestingConsensusContext(t, 3, NO_ATTACK)
    proposedValue := "42"
    currentNodeId := ConsensusContext.NodeId

    sendMvcMessagesOf2CorrectNodes()

    addMvcMessage(currentNodeId, VBB_INIT, BRB_INIT,
         currentNodeId, proposedValue)

    // transient error - inject invalid valid-init message
    addMvcMessage(currentNodeId, VBB_VALID, BRB_INIT,
        currentNodeId, "false")

    go MultiValueConsensus(proposedValue)

    sendBcMessagesOf2CorrectNodes()

    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.4 Combination of Byzantine Attacks and transient errors

After looking at the two group scenarios above, it is interesting to evaluate if the algorithm can outstand and recover when transient errors and Byzantine attacks happen simultaneously. We repeat that if a transient error occurs in a node that is not self-stabilizing, its behavior can combine with existing Byzantine nodes and, in conclusion, not allow the system to reach consensus. Below we show some basic combination scenarios.

5.4.1 Byzantine node that is idle + Transient error on the proposed value

In the first combination test, we combine the Byzantine Idle attack shown in Section 5.2.1 with the transient error on the INIT message case. We will see that, in this case, consensus is still reached.

```go
func TestMultiValueConsensusWithByzantineIdleAndTransientErrorOnInit(t
*testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 6, NO_ATTACK)
    proposedValue := "42"

    sendMvcMessagesOf4CorrectNodes()

    go MultiValueConsensus(proposedValue)

    // transient error - inject invalid init message
    addMvcMessage(currentNodeId, VBB_INIT, BRB_INIT,
      currentNodeId, "1231234")

    sendBcMessagesOf4CorrectNodes()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.4.2 Byzantine node with invalid proposal + Transient Error on the proposed value

Another combination of failures that, in a different case, would not let the system reach consensus would be a Byzantine node proposing an invalid value. By chance, a transient error happens in a correct node that makes its proposed value the same as the Byzantine one. If we assumed that we reached the maximum number of Byzantines allowed in the system to work correctly, we could come to a state where we have another one that agrees with the Byzantines. We test this combination in the following test, and we can again see that consensus has been reached. Only one Byzantine node is tolerable, proposing the value *7*, and at the same time, a transient error changes a correct node's proposal to *7* as well.

```go
func TestMultiValueConsensusWithByzantineInvalidValueAndTransient
ErrorOnINIT(t *testing.T) {

    // sets up testing environment with total number of nodes
    // and attack scenario that should be performed by current node
    setupTestingConsensusContext(t, 6, NO_ATTACK)
    proposedValue := "42"

    sendMvcMessagesOf4CorrectAndOneFaultyNode()

    go MultiValueConsensus(proposedValue)

    // transient error - overwrite init message
    addMvcMessage(currentNodeId, VBB_INIT,
        BRB_INIT, currentNodeId, "7")

    sendBcMessagesOf4CorrectAndOneFaultyNode()

    // keep getting mvc result until consensus is reached
    var mvcResult string
    for {

        mvcResult = MvcResult()
        if mvcResult != NonDecidedValue {
            break
        }
    }

    // check that the expected value is decided
    assert.Equal(t, mvcResult, proposedValue)
}
```

5.5 Conclusion

In this chapter, we presented some of our studied scenarios in which the algorithm is designed to perform without issues. We saw that even in the combination of Byzantine Attacks and Transient errors, the system could recover and reach consensus as long as the code and system variables were intact, and the number of Byzantine nodes was less than a third of the total number of nodes. To our knowledge, no combination of scenarios like the above could prevent the system from working as it was designed, always considering the type of allowed errors and attacks and the minimum requirements.

Chapter 6

Experimental Analysis

6.1 Experimental Environment

We performed experiments on a local conventional workstation for the experimental analysis and then deployed the algorithm on a cluster reserved on Cloudlab [33]. The goal was to first validate the algorithm towards the theoretical assumptions on the local workstation and then move it to a distributed-like system for more precise and scalable measurements. In both environments, we started multiple MVC nodes per machine, where each node printed the decided value, the time until consensus was reached, and message size and count complexities. Based on these, we could calculate results on the three following properties.

1. **Operation latency**: The average time needed by the system to reach consensus. Specifically, it is the time difference between the start of nodes' execution and the exact time a value is decided.
2. **Message Complexity**: The average number of messages broadcasted by the nodes until consensus is reached.
3. **Bit Complexity**: The average payload size sent by nodes until consensus is reached.

Regarding Message and Bit Complexities, by recalling Section 3.3.2.1, nodes group all their messages and send them as one big message. Moreover, the same message can be repeatedly sent hundreds of times without receiving any proof of delivery. Based on the above, *Message Complexity* and *Bit Complexity* count the number and the size of the big messages, e.g., if 2 big messages are sent containing 16 algorithms (VBB, BRB) messages, then *message complexity* is 2.

Finally, we will observe how the three properties described above fluctuate while increasing the number of total nodes to verify the algorithm's scalability.

6.1.1 Local Workstation Environment

At first, we executed various experiments on an ordinary personal computer, with varying nodes and the occurrence of a transient error, to verify the algorithm's validity and theoretical specifications. The local machine contains the specifications shown in Figure 26.

Operating System	RAM	CPU(s)	Thread(s) per core	Core(s) per socket	Socket(s)	CPU Model name
Linux Ubuntu 20.04 LTS x86_64	DDR4 8GB	8	2	4	1	Intel(R) Core(TM) i7-7700HQ CPU

Figure 26: Local workstation environment specifications

6.1.2 Cloudlab Environment

After local experiments, we were granted access to the Cloudlab testbed platform to emulate distributed systems' executions. There are many machine setups and configurations, but we used ten machines of the type *xl170* [41] and started multiple nodes per machine. The cluster machines' specifications are shown in Figure 27.

Node ID	Operating System	RAM	CPU(s)	Thread(s) per core	Core(s) per socket	Socket(s)	CPU Model name
0	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
1	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
3	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
4	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
5	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
6	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
7	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
8	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
9	Linux Ubuntu 18.04 LTS x86_64	DDR4 8GB	20	2	10	2	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz

Figure 27: The Cloudlab setup specifications

6.2 Experiment Scenarios

We executed various scenarios that combined Byzantine nodes' presence and transient errors.

As a transient error, for simplicity, we delete the INIT proposed message of the node before proposing to Binary Consensus, leading to the deletion of all messages in the node due to consistency tests. The maximum tolerant number of Byzantine nodes is used in the scenarios where Byzantine nodes are used.

In the following sub-sections, we describe the different Attack Scenarios that Byzantine nodes perform.

6.2.1 Failure-free Scenario

The first scenario is the basic scenario where there are no Byzantine nodes or arbitrary-transient faults. In this scenario, every node is non-faulty and acts as expected, with no probability of arbitrary-transient faults. The results of this scenario will be used as the reference point for the rest of the scenarios.

6.2.2 Byzantines Idle Scenario

During this scenario, every Byzantine node stays idle, meaning it does not take part in reaching a consensus. Simply, it does not send or reply to any of the messages. The Byzantine nodes can still keep executing and receiving messages, but they give no signs of execution to the rest of the nodes. This basic scenario is crucial, as it emulates crashed/refusing to work at all nodes in a system that still manages to perform.

6.2.3 Byzantines Half & Half Attack Scenario

The Half & Half Attack Scenario is the first of the two scenarios where Byzantines try to sabotage consensus by sending invalid messages. In this scenario, Byzantine nodes reply and broadcast all messages, but they send invalid messages to half of the nodes. Specifically, when a message (BRB or VBB) is sent, the correct messages are sent to *n/2*, and a set of messages with invalid/modified values is sent to the other *n/2*. This scenario helps us see the overhead added in the presence of Byzantine nodes, although consensus is still reached.

6.2.4 Byzantines Random Messages Attack Scenario

Similar to the previous attack scenario, in this scenario, Byzantine nodes try to sabotage consensus by sending invalid messages with random values at any time. In plain, a Byzantine node, before sending a message, modifies its value with a randomly generated value and then sends it. This attack also adds overhead, and in specific cases, it is even more significant than the Half & Half attack since Byzantine nodes sent invalid values to all other nodes. This attack is a good scenario put to the test to see the results when Byzantine nodes send nothing coordinated but invalid random values.

6.3 Results and Theoretical Analysis Evaluation

In this section, we are first going to look at the theoretical analysis from the paper [6], then analyze each setup and the executed scenario alongside their results and finally compare them to the theoretical analysis.

6.3.1 Theoretical Specifications and Performance

First, we evaluate whether the algorithm has optimal resilience by assuming $t < n/3$, where t is the number of faulty nodes and n is the number of total nodes. This assumption is already verified in the scenarios we covered in Chapter 5, where we saw that consensus is reached when the number of faulty nodes is not more than a third of the total of nodes. Here, we will verify that the same happens with more nodes in real-life-like simulations. Additionally, we will execute scenarios where arbitrary-transient errors are injected in one node and how this affects the overall system. This specification was again covered in Chapter 5, but we will verify it through our experiments. Lastly, we will evaluate if the self-stabilizing algorithm performs similarly to the non-self-stabilizing variation, with only a small expected overhead.

6.3.2 Scenarios, setups, and results

On the local workstation presented in Section 6.1.1, we run experiments to calculate the *operation latency, message complexity,* and *bit complexity* over the increasing number of total nodes *n*, using Byzantine Attacks and injecting transient errors. We executed scenarios using 3 to 15 nodes to validate the algorithm's behavior in basic setups.

On the Cloudlab cluster, we executed specific setups to observe the algorithm's performance over scalable systems. Precisely, we executed the Failure-free Scenario using an increasing number of nodes per machine, starting from 1 node per machine (*total: 10 nodes*) to 15 nodes per machine (*total:150 nodes*). We then kept the number of nodes per machine stable, precisely 10 per machine (*total: 100 nodes*), and executed scenarios combining Byzantine nodes and transient errors.

Finally, again on Cloudlab, we executed more basic scenarios with 3 up to 20 nodes, with the Byzantine Idle and Byzantine Half & Half Attacks, to compare the studied algorithm to the non-self-stabilizing version by Mostéfaoui *et al.* [3], using the implementation by Petrou [7].

6.3.2.1 Operation Latency

In the figures in this section, we look at how *operation latency* changes on scenarios executed in the two environments and setups explained above.

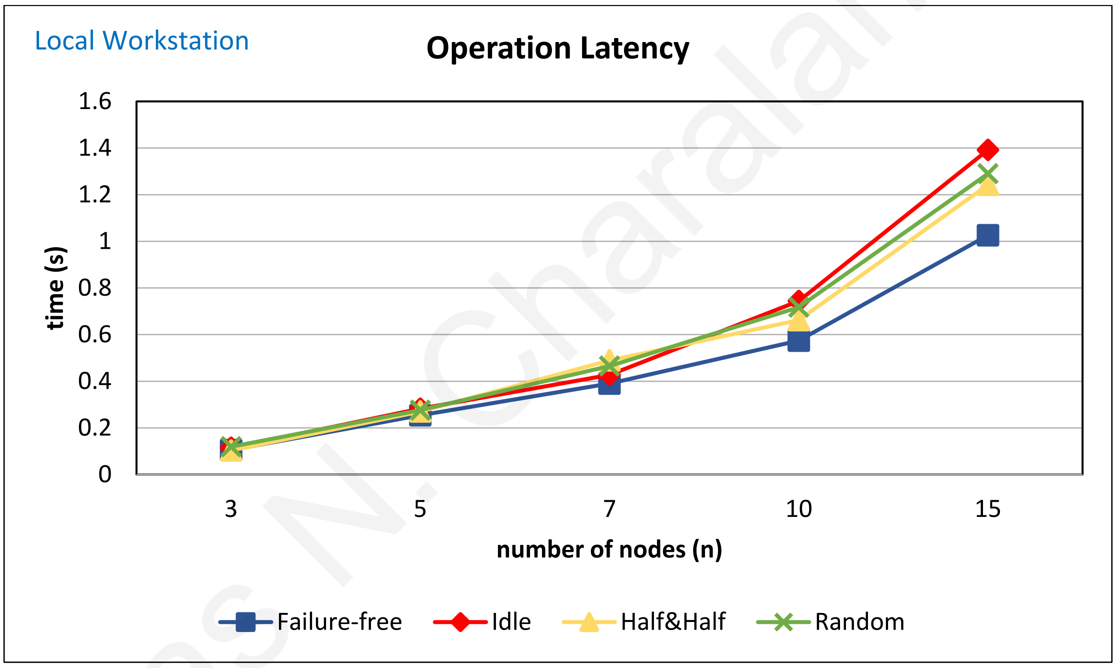

Figure 28: Operation Latency (in seconds) on Byzantine Attack Scenarios on Local Workstation

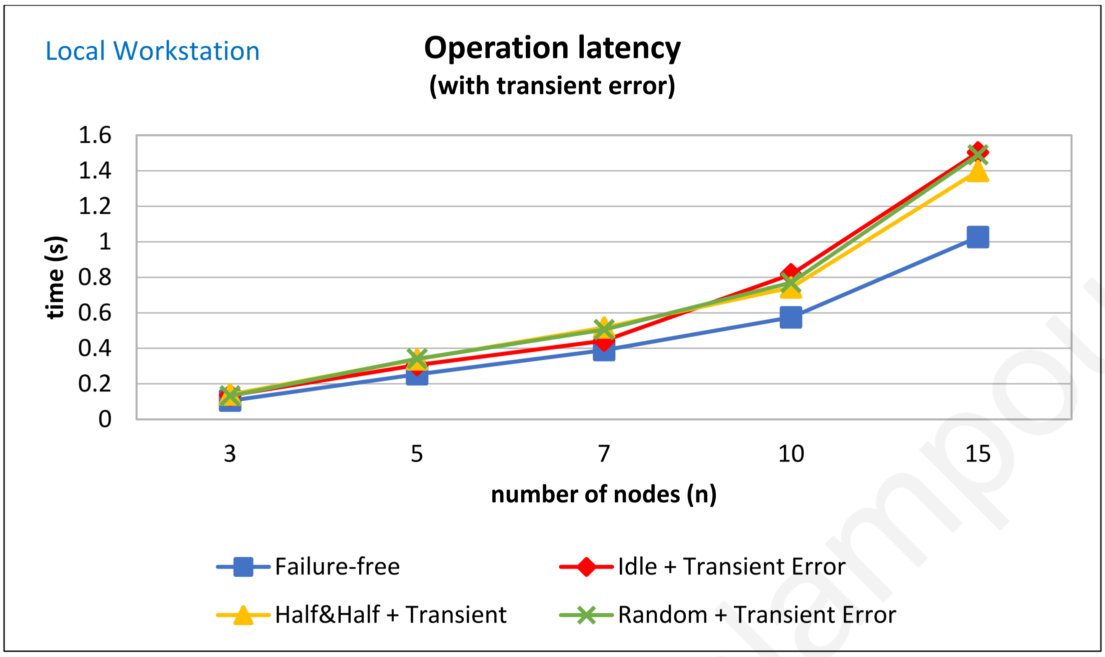

Figure 29: Operation Latency (in seconds) on Byzantine Attack Scenarios, combined with transient errors, on Local Workstation

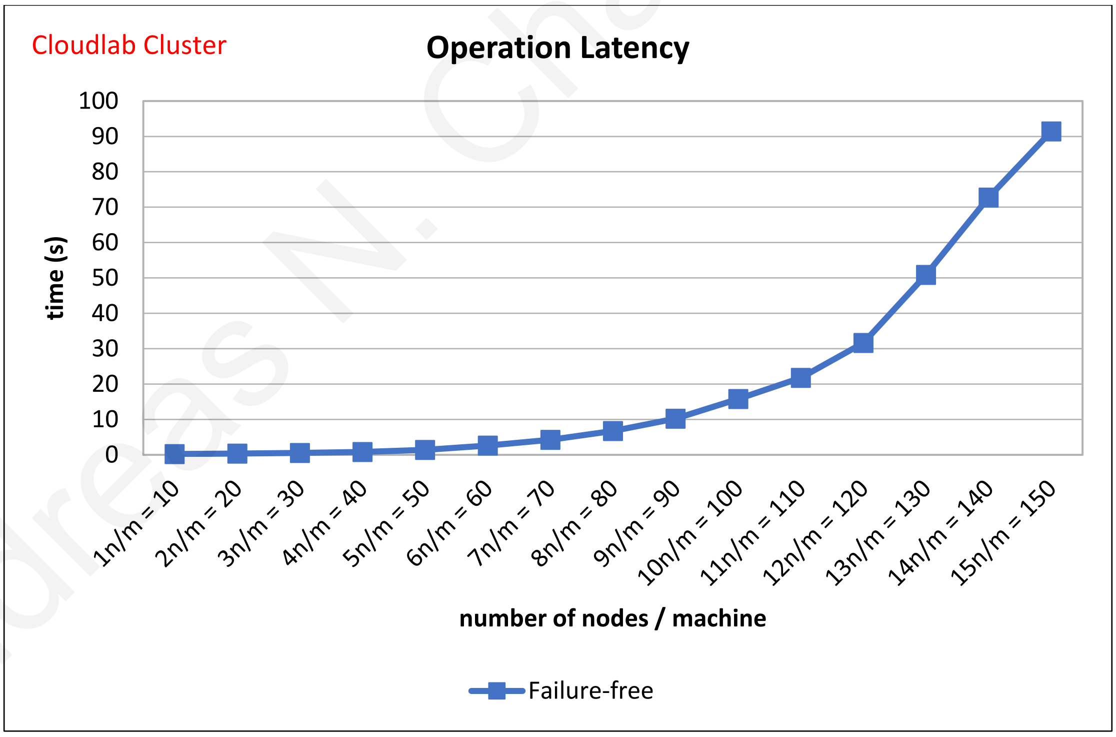

Figure 30: Operation Latency (in seconds) on scaling number of nodes per machine on Cloudlab Cluster

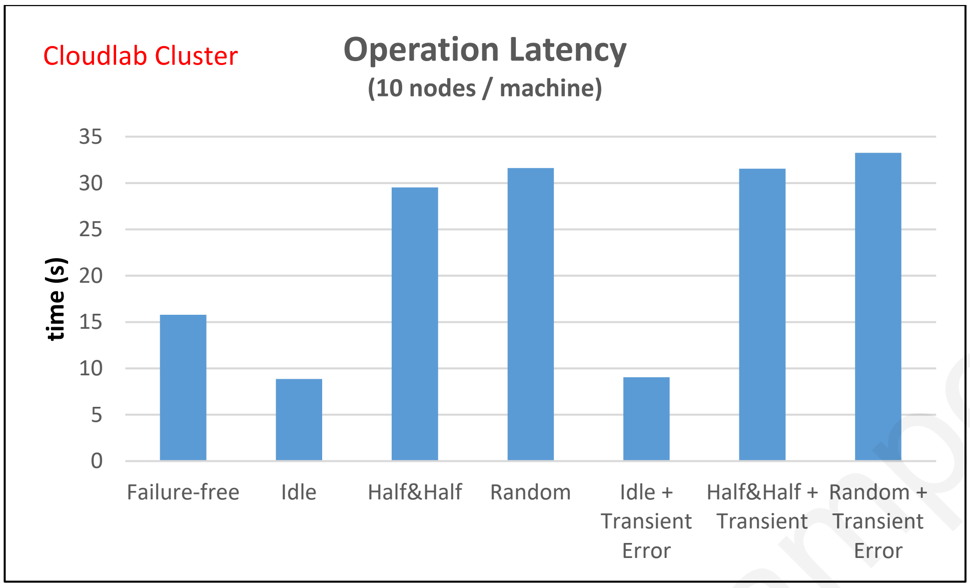

Figure 31: Operation Latency with Byzantine Attacks and Transient Errors, on Cloudlab Cluster (10 nodes per machine)

Based on the life-like simulation results above, combined with the test cases in Chapter 5, we can verify that the algorithm is resilient for up to $t < n/3$ Byzantine nodes. This conclusion comes from using the maximum tolerant number of Byzantine nodes in every Byzantine Attack Scenario, and the consensus was still reached. Also, we see that the algorithm is self-stabilizing as it reaches consensus in the scenarios where transient faults are injected, even when there are maximum tolerant Byzantine nodes in the system.

In terms of *operation latency,* initially, we can see an increase in the time needed to reach consensus when the transient fault is injected, especially in the presence of Byzantine nodes. The results indicate that cases of transient errors are detected instantly, and additional "graceful" time is needed to reach a consensus.

Additionally, we can observe how needed time is changed in different Byzantine Attack Scenarios. Starting with the Byzantines Idle Attack Scenario, we notice a minor increase in the *operation latency* compared to the Failure-free Scenario because, in the latter, each node had more available nodes to receive messages from in order to proceed. We see later in scalable scenarios that this attack decreases the time needed in cases where the system is flooded with resources and communication overhead since this will decrease it. In the local workstation setup, we also see our algorithm implementation handles Half&Half and Random Message attacks with similar drawbacks.

The Cloudlab cluster results show that the algorithm is scalable and works for many nodes, as long as the number of Byzantine nodes is not greater than a third of the total number of nodes. We can again see in every scenario the additional small overhead added when a transient error occurs. It is also noticeable, as already said, how the Idle Byzantine Attack Scenario decreases the *operation latency* in more significant node numbers since there is less resource utilization. Similar to the local workstation results, the Half & Half and Random Attacks have similar drawbacks.

6.3.2.2 Messages Complexity

In this section, we are evaluating the message complexity of the algorithm. As already described, the message complexity shows the average number of messages sent per node. Similar to the section above, the following figures show executions on both the local workstation and the Cloudlab cluster.

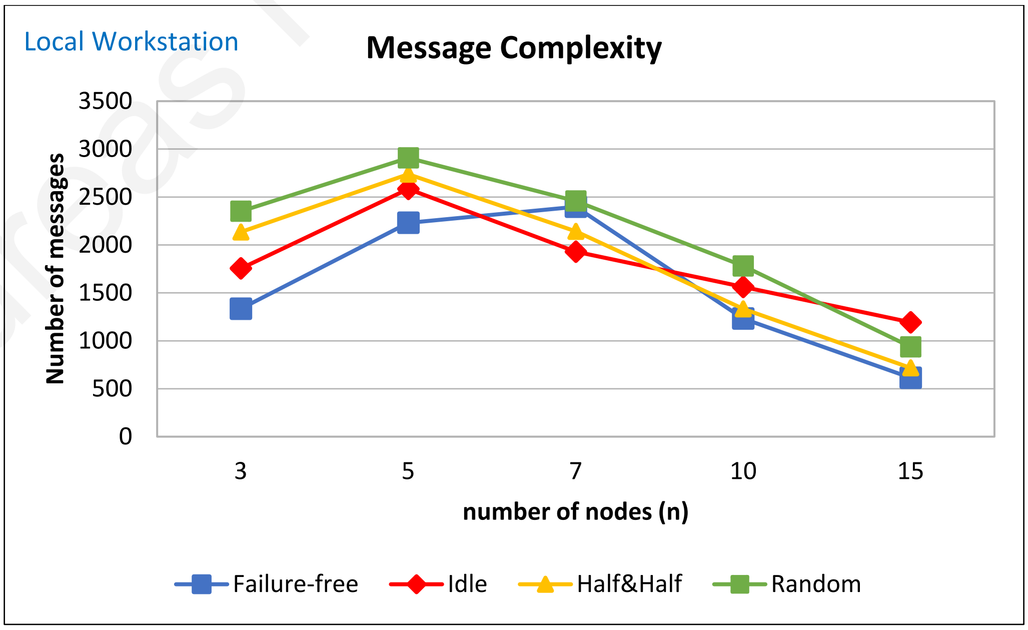

Figure 32: Message Complexity on Byzantine Attack Scenarios on Local Workstation

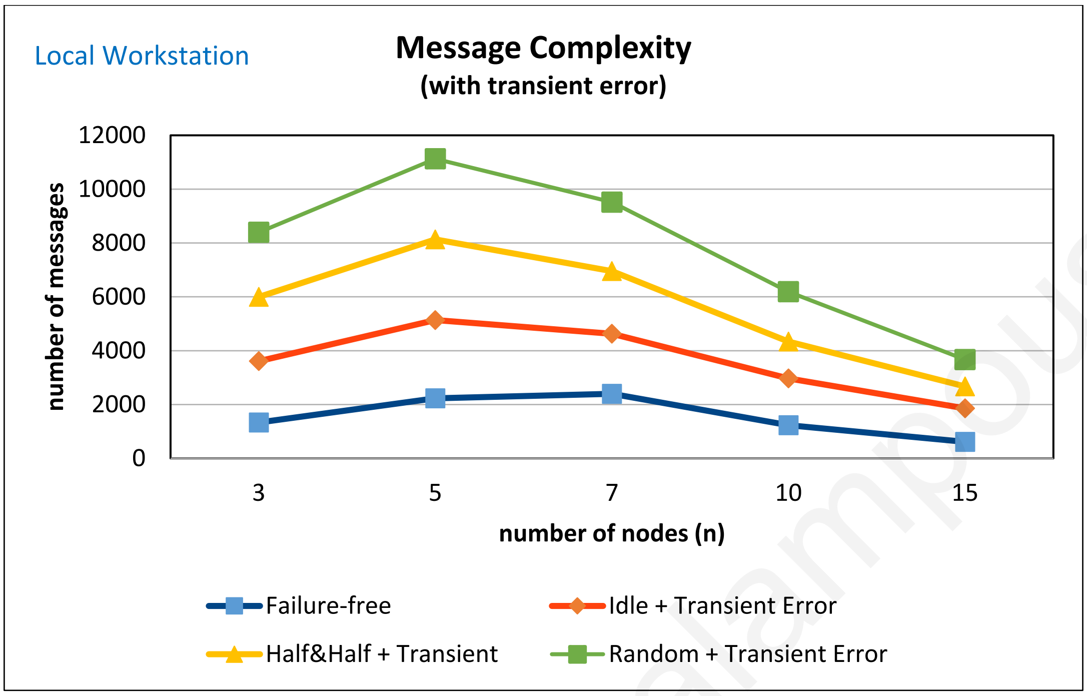

Figure 33: Message Complexity on Byzantine Attack Scenarios, combined with transient errors, on Local Workstation

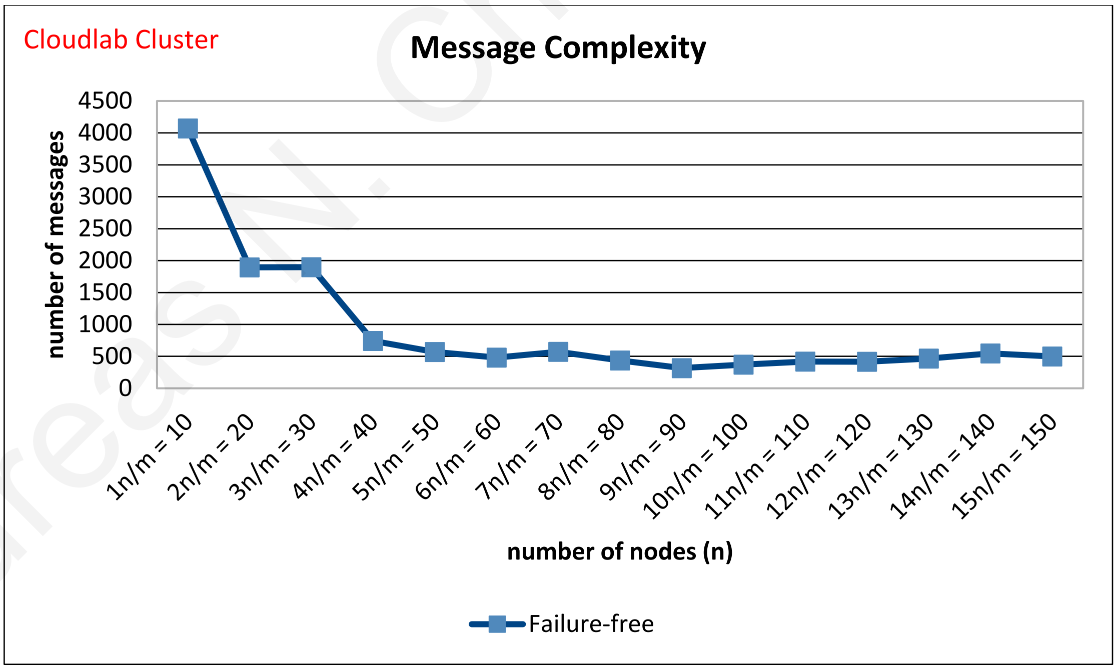

Figure 34: Message Complexity on scaling number of nodes per machine on Cloudlab Cluster

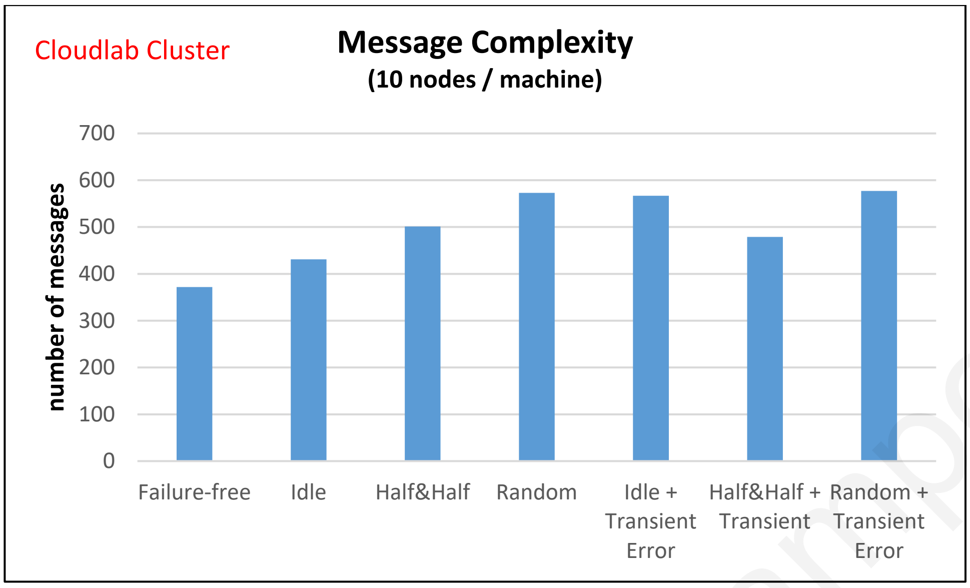

Figure 35: Message Complexity with Byzantine Attacks and Transient Errors, on Cloudlab Cluster (10 nodes per machine)

Due to the algorithm’s design, the for-loop, and the continuous broadcasting of messages, resources are expected to be exploited when they are available. We see that on both machines, in initial scenarios where the number of nodes is small and the number of resources more significant, there is a massive number of messages sent, even though the consensus is reached fast enough. We see that message complexity decreases as nodes number increases until it becomes stable at a specific point.

In terms of faults and how they change the message complexity, we see that when a transient error occurs, the number of messages is significantly increased for the smaller number of nodes. The message complexity at the 15-node setup is closer to the transient-error-free scenarios. In the local workstation executions, we see that the transient error has more impact on the message complexity. However, on the Cloudlab cluster, we see the impact to be much less since, in scaled setups, a small transient error will lead to a much less percentage of sent messages needed to reach consensus. We can conclude that Byzantine faults, in combination with transient errors, can add overhead to message complexity depending on resource availability.

6.3.2.3 Bit Complexity

Next, we take a look at the bit complexity, which shows the average payload size sent per node. Since Bit Complexity and Message Complexity have similar behaviors, we can see how Byzantine attacks also increase Bit Complexity.

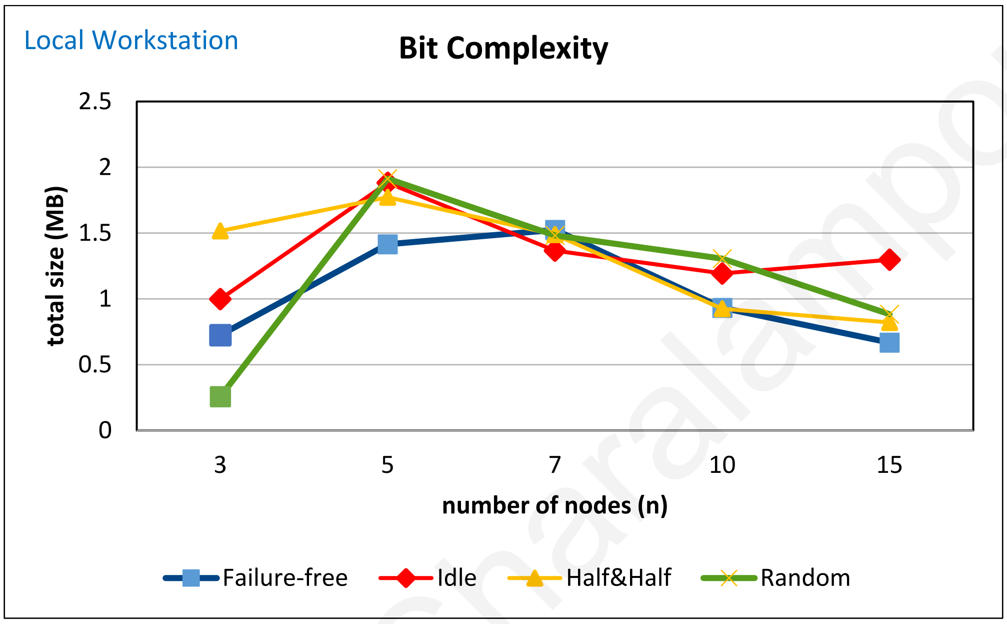

Figure 36: Bit complexity (in MB) on Byzantine Attack Scenarios on Local Workstation

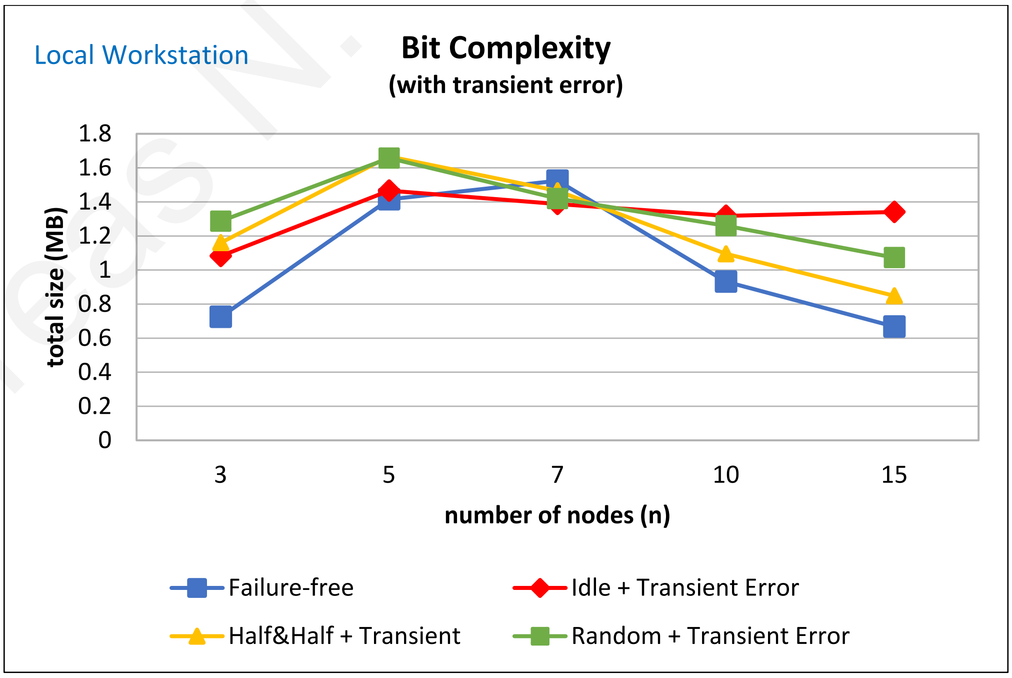

Figure 37: Bit Complexity (in MB) on Byzantine Attack Scenarios, combined with transient errors, on Local Workstation

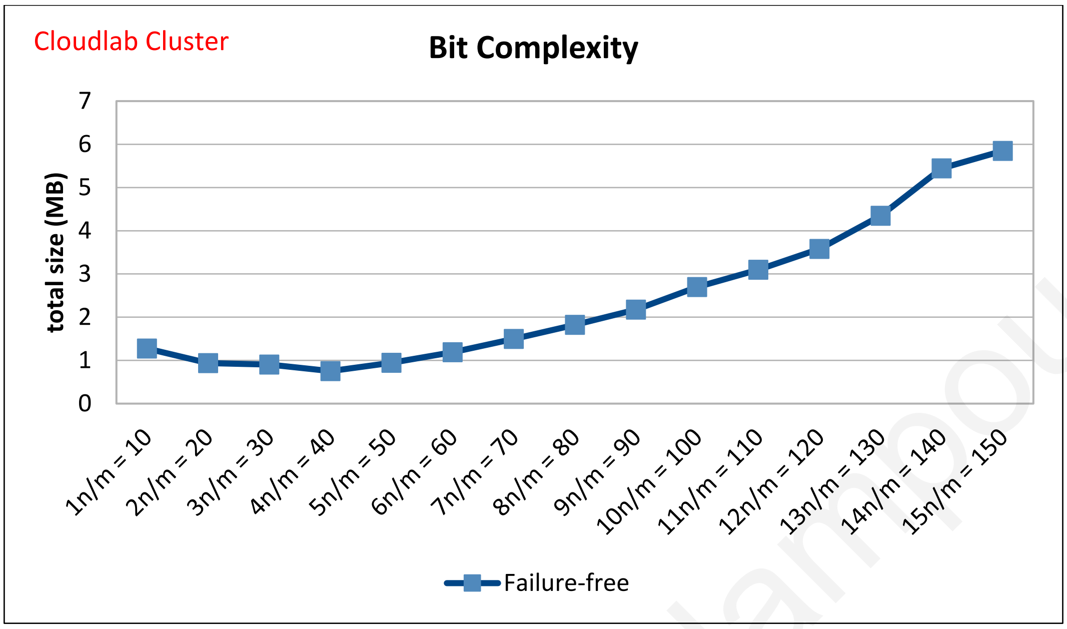

Figure 38: Bit Complexity (in MB) on scaling number of nodes per machine on Cloudlab Cluster

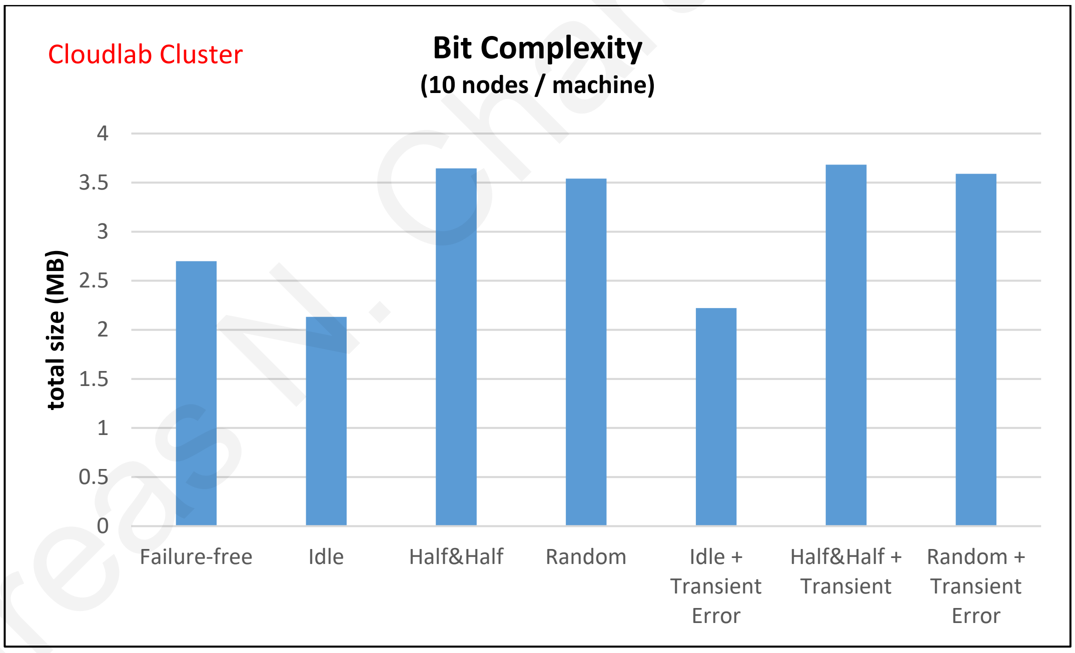

Figure 39: Bit Complexity (in MB) with Byzantine Attacks and Transient Errors on Cloudlab Cluster (10 nodes per machine)

We see more scattered results in the local workstation, making it hard to extract conclusions, in contrast to Message Complexity. The reason is that with the existing checks, we cannot know what messages were repeatedly broadcasted and how big they were. In cases where small messages were repeatedly sent, there should only be a slight increase of bit complexity. The same applies the other way around.

Although the above, we can still extract conclusions from the scaled scenarios in the Cloudlab Cluster. In the equivalent figures, we can see how bit complexity is decreased until 4 nodes per machine, and then it has a standard increase. As expected, we see an increase when Byzantine faults happen, especially when combined with transient errors. We also see how bit complexity is decreased during Idle Byzantine Attack, where fewer resources are used.

6.3.2.4 Comparison with the non-self-stabilizing algorithm

In this part, we compare our implementation to the non-self-stabilizing algorithm by Mostéfaoui *et al.* [3], using the implementation by Petrou [7]. As already said, the two implementations use the same technology stack, built using Go and ZeroMQ. For the operation latency comparison, we executed scenarios of 3 to 20 nodes on the Cloudlab Cluster, and for the message complexities, we executed scenarios locally with 3 to 15 nodes. We calculated the operation latency, message, and bit complexity in failure-free scenarios and Byzantine Idle and Half & Half Attacks.

Of course, one advantage of the studied algorithm, and as its name implies, is that it can recover from arbitrary-transient errors, while the algorithm by Mostéfaoui cannot. This capability is expected to increase the operation latency compared to the non-self-stabilizing algorithm since many checks are performed. The goal is to keep this increase as little as possible.

The figures below show the performance difference between the two algorithms.

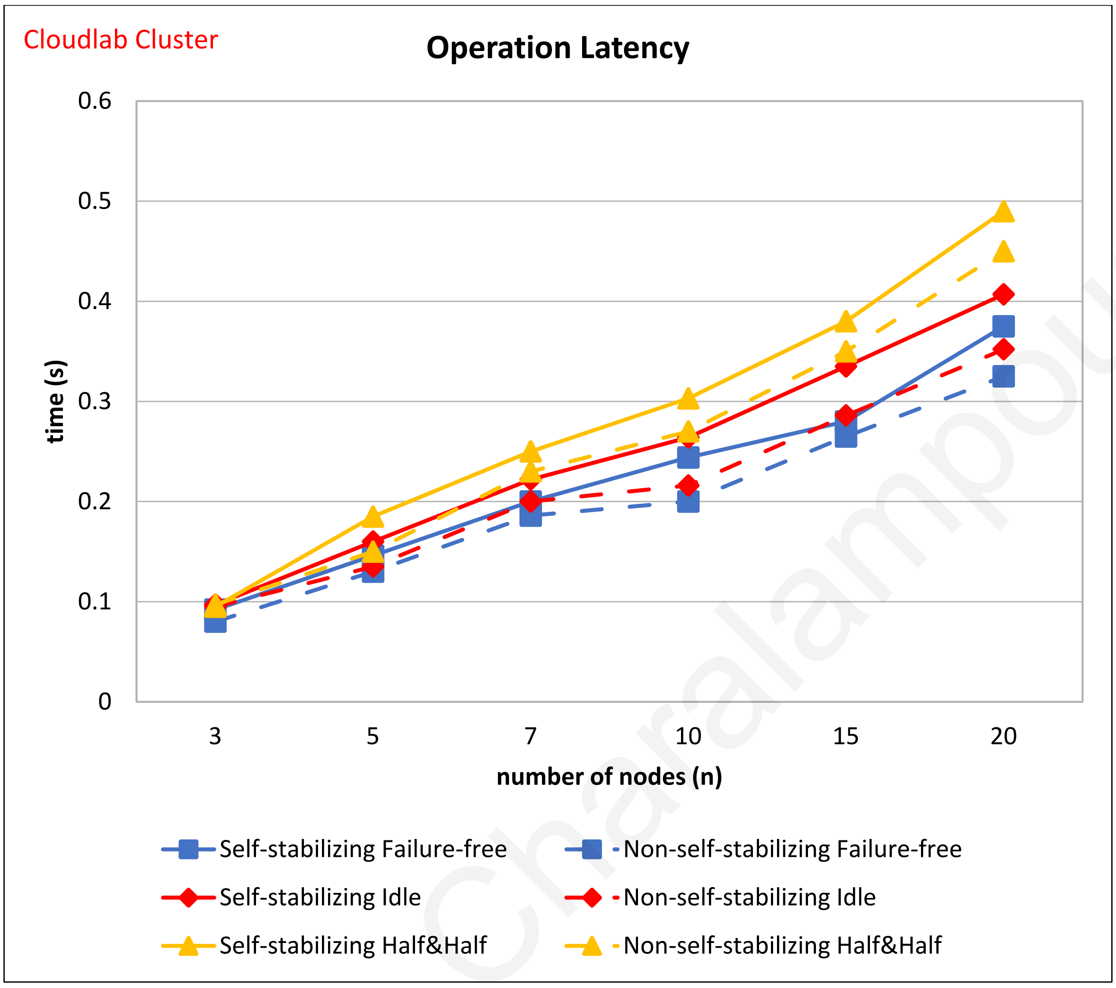

Figure 40: Operation Latency (in seconds) comparing Self-stabilizing and Non-self-stabilizing variants using Byzantine Attacks on Cloudlab Cluster

Starting with the operation latency, we see how it increases in the self-stabilizing version. The two algorithms behave similarly in every scenario, with the operation latency increasing at a similar rate while the number of nodes increases. In conclusion, the performance drawback in exchange for the self-stabilization property is graceful and therefore acceptable.

Finally, we empirically compare Message and Bit Complexity and how they change depending on the scenario. For these properties, we cannot be exact in comparing because of the different designs of the two algorithms and how they measure messages. We, therefore, compared them according to how they fluctuate. Below we can see the measurements of the non-self-stabilizing algorithm as they were extracted from Petrou [7].

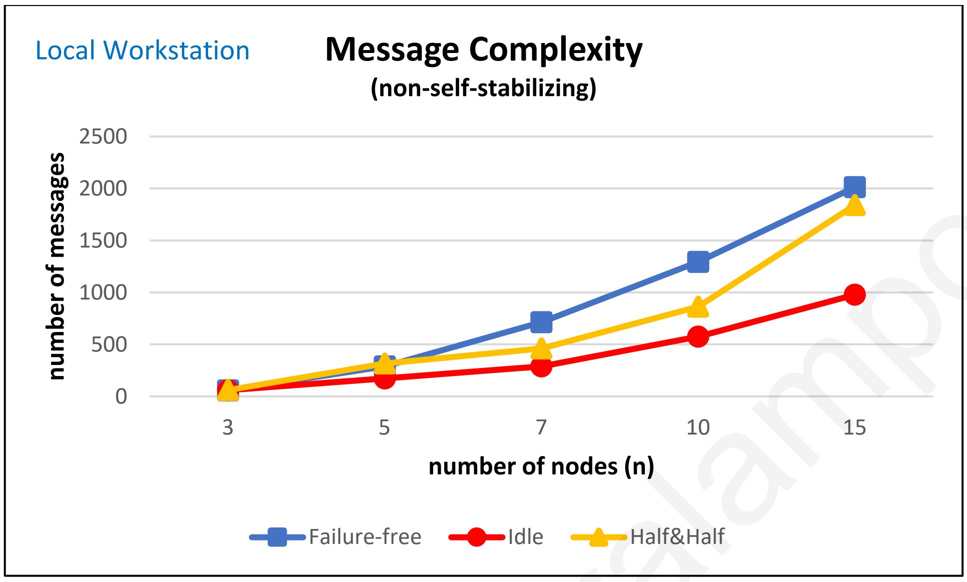

Figure 41: Message complexity of the non-self-stabilizing algorithm on Local Workstation

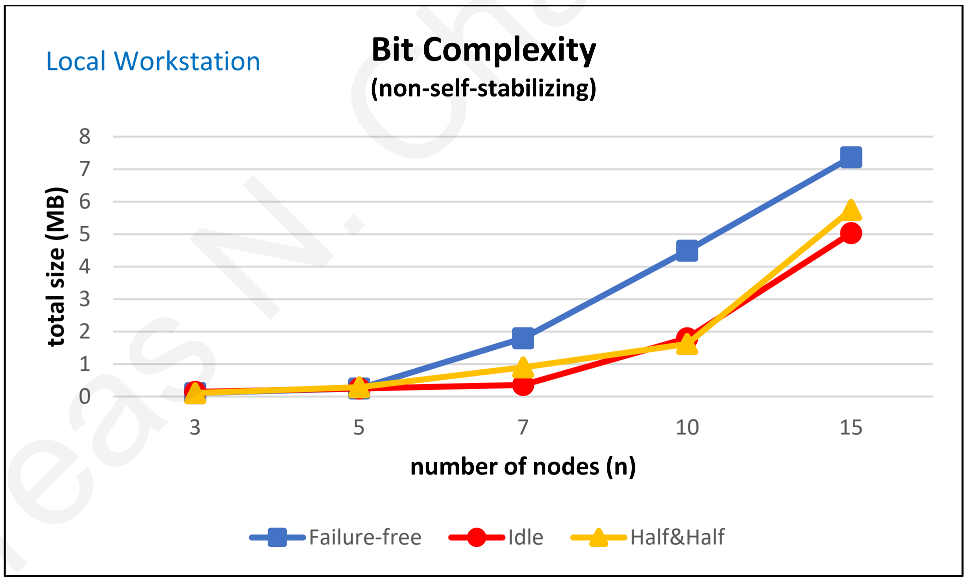

Figure 42: Bit complexity (in MB) of the non-self-stabilizing algorithm on Local Workstation

Comparing the figures above with those in Sections 6.3.2.2 and 6.3.2.3, we can again verify the exploitation of resources happening in the self-stabilizing algorithm, as the Message and Bit Complexities are more significant for small setups and decrease until a point. In contrast, the non-self-stabilizing algorithm starts with more minor complexities and gradually increases.

6.4 Experimental Summary

This chapter evaluated the algorithm’s specifications, validity, and properties. We saw how operation latency, message, and bit complexity change in different scenarios, including Byzantine faults in combination with arbitrary-transient-faults.

First, we saw with real-life scenarios that the studied algorithm is indeed tolerant for up *to t* < *n/3* faulty nodes, no matter the attack they perform. We verified once again that the algorithm is self-stabilizing and can detect and recover from transient faults. Both Byzantine and transient faults add an overhead to the time needed for reaching consensus, but the algorithm can still perform with acceptable results.

With the scaled setup executions on the Cloudlab cluster, we verified that the algorithm is scalable and can perform well under huge setups with hundreds of nodes.

Depending on its design, one big drawback of the algorithm is that an enormous number of messages are repeatedly sent, leading to the exploitation of resources. Additional studies can be made to find the perfect settings for broadcasting to get the most out of the algorithm.

Another essential outcome was the comparison of the algorithm with the non-self-stabilizing algorithm by Mostéfaoui *et al.* [3]. We saw that the studied algorithm could perform similarly to the non-self-stabilizing variation, with minimal additional overhead.

Chapter 7

Conclusion

7.1 Summary

In this dissertation, we looked at essential properties that a distributed system must have, such as Fault Tolerance, Self-stabilization, and their combination towards consensus in the presence of Byzantine nodes and arbitrary transient errors.

During the thesis, we analyzed the algorithm by Duvignau et al. [6], the first proposed Byzantine and intrusion-tolerant Self-stabilizing multivalued consensus algorithm. We inspected the building blocks needed for the implementation, one by one, and the challenges and solutions for the algorithm to behave as a Self-stabilizing algorithm.

After studying the algorithm, we built, to the best of our knowledge, the first implementation of the algorithm [6] by using the Go programming language with the ZeroMQ messaging library, which can be integrated into other algorithm stacks. The algorithm was then proved to handle complicated faults, including Byzantine and transient faults combined.

We evaluated the algorithm by executing real-life scenarios on a local workstation and a cluster using Emulab/Cloudlab testbed platform. We evaluated the algorithm's theoretical properties and performance through various setups and scenarios. We executed additional scenarios that would help us demonstrate that the algorithm can perform well on scalable systems. Finally, we compared our implementation to a Non-self-stabilizing variation [3] [7] and verified the self-stabilizing overhead, which is nevertheless minor and acceptable. We were able to discover and point out drawbacks/limitations, mainly on resource exploitation, and how additional studies can improve our implementation.

7.2 Future Work

Implementing the algorithm is only the beginning of what can be achieved later. As stated in Chapter 3, the algorithm stack contains building blocks on top of our implementation. Another set of protocols can be added for more reliable results, like Reliable Broadcast with total-order delivery and Emulation of state-machine replication.

Also, in our implementation, to get the best result, we do not have any timing mechanism in the main for-loop, leading to exploitation of the system resources since the for-loop runs indefinitely, pushing messages all the time. A possible solution would be to add a simple "*sleep*" in each iteration or avoid sending messages in every iteration, for example, every ten iterations. This way, the sent messages are not much more than the number needed, leading to a performance increase. Additional study can be done to find the most suitable tweaks needed for the best result.

Another work that could be done, but at this time-being and resources was not feasible, is running the algorithm on a complete Distributed System, such as Amazon Web Service (AWS), with nodes scattered over long distances and different continents. Theoretically, our implementation should be able to perform on such a setup, and the only drawback should be the time needed for consensus to be reached.

7.3 Personal Retrospection

I am again grateful that I had the chance to work on a complete project of this item. By getting hands-on with existing functional projects, this thesis allowed me to understand Distributed Systems, Byzantine Fault Tolerance, Consensus, and others. Additionally, I had the opportunity to explore Self-stabilization, understand why and how it is vital to Distributed Systems, and finally become confident in implementing one such system. Moreover, I learned an entirely new language, Go, from scratch and familiarized myself with the ZeroMQ library through this project. In general, I had the chance to gain more experience by implementing this project from the beginning.

Bibliography

[1] "Wikipedia - Distributed Computing," [Online]. Available: https://en.wikipedia.org/wiki/Distributed_computing. [Accessed April 2022].

[2] M. H. Degroot, "Reaching a Consensus," *Journal of the American Statistical association,* vol. 69, no. 345, pp. 118-121, 1974.

[3] A. Mostéfaoui, M. Hamouma and M. Raynal, "Signature-free asynchronous Byzantine consensus with t < n/3 and O (n2) messages," *Proceedings of the 2014 ACM symposium on Principles of distributed computing,* pp. 2-9, 2014.

[4] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *Concurrency: the Works of Leslie Lamport,* vol. 203, no. 226, pp. 203-226, 2019.

[5] S. Dolev, R. I. Kat and E. M. Schiller, "When consensus meets self-stabilization," *Journal of Computer and System Sciences,* vol. 76, no. 8, pp. 884-900, 2010.

[6] R. Duvignau, M. Raynal and E. M. Schiller, "Self-stabilizing Byzantine-and Intrusion-tolerant Consensus," *arXiv preprint arXiv:2110.08592,* 2021.

[7] V. Petrou, "Implementation and Evaluation of a Randomized Byzantine Fault Tolerant Distributed Algorithm," May, 2021.

[8] "Golang Documentation," [Online]. Available: go.dev/doc. [Accessed April 2022].

[9] "ZeroMQ," [Online]. Available: https://zeromq.org/get-started/. [Accessed April 2022].

[10] "Emulab," [Online]. Available: https://www.emulab.net/. [Accessed June 2022].

[11] A. Sari and M. Akkaya, "Fault Tolerance Mechanisms in Distributed Systems," *International Journal of Communications, Network and System Sciences,* vol. 08, no. 12, p. 12, 2015.

[12] A. Kumar, Y. S. Rama and R. J. Anjali, "Fault tolerance in real time distributed system," *International Journal on Computer Science and Engineering,* vol. 3, no. 2, pp. 933-939, 2011.

[13] M. J. Fisher, N. A. Lynch and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM),* vol. 32, no. 2, pp. 374-382, 1985.

[14] M. K. Aguilera, S. Toueg and B. Deianov, "Revisiting the weakest failure detector for uniform reliable broadcast," *International Symposium on Distributed Computing,* pp. 19-34, 1999.

[15] S. Toueg, "Randomized byzantine agreements," *Proceedings of the third annual ACM symposium on Principles of distributed computing,* pp. 163-178, 1984.

[16] C. Georgiou, I. Marcoulis, M. Raynal and E. M. Schiller, "Loosely-self-stabilizing Byzantine-tolerant binary consensus for signature-free message-passing systems," *International Conference on Networked Systems,* no. 36-53, 2021.

[17] Y. Yoshida, "Toyota case: Single bit flip that killed, 2013," [Online]. Available: https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/. [Accessed May 2022].

[18] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM,* vol. 17, no. 11, pp. 643-644, 1974.

[19] M. Raynal, "Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach," *Springer,* 2018.

[20] P. Blanchard, S. Dolev, J. Beauquier and S. Delaët, "Practically self-stabilizing Paxos replicated state-machine," *International Conference on Networked Systems,* pp. 99-121, 2014.

[21] O. Lundström, M. Raynal and E. M. Schiller, "Self-stabilizing multivalued consensus in asyncrhonous crash-prone systems," *2021 17th European Dependable Computing Conference (EDCC),* pp. 111-118, 2021.

[22] O. Lundström, M. Raynal and E. M. Schiller, "Self-stabilizing indulgent zero-degrading binary consensus," *International Conference on Distributed Computing and Networking 2021,* pp. 106-115, 2021.

[23] M. Ben-Or, B. Kelmer and T. Rabin, "Asynchronous secure computations with optimal resilience," *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing,* pp. 183-192, 1994.

[24] A. Mostéfaoui and M. Raynal, "Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes," *IEEE Transactions on Parallel and Distributed Systems,* vol. 27, no. 4, pp. 1085-1098, 2015.

[25] A. Mostéfaoui and M. Raynal, "Signature-free asynchonous Byzantine systems: from multivalued to binary consensus with t < n/3, o(n^2) messages, and constant time," *Acta Informatica,* vol. 54, no. 5, pp. 501-520, 2017.

[26] M. Correia, N. F. Neves and P. Veríssimo, "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures," *The Computer Journal,* vol. 49, no. 1, pp. 82-96, 2006.

[27] N. F. Neves, M. Correia and P. Veríssimo, "Solving vector consensus with a wormhole," *IEEE Transactions on Parallel Distributed Systems,* vol. 16, no. 12, pp. 1120-1131, 2005.

[28] A. Mostéfaoui, M. Hammouma and R. Michel, "Signature-free asynchronous Byzantine consensus with t < n/3 and o(n^2) messages," *ACM Principles of Distributed Computing,* 2014.

[29] A. Mostéfaoui, M. Hamouma and M. Ryanal, "Signature-free asynchronous binary Byzantine consensus with t < n/3, o(n^2) messages, and O(1) expected time," *Journal of the ACM (JACM),* vol. 62, no. 4, pp. 1-21, 2015.

[30] "Wikipedia - Go Programming Language," [Online]. Available: https://en.wikipedia.org/wiki/Go_(programming_language). [Accessed April 2022].

[31] "freecodecamp - What is Go? Golang Programming Language Meaning Explained," [Online]. Available: freecodecamp.org/news/what-is-go-programming-language. [Accessed April 2022].

[32] "The Emulab Manual," [Online]. Available: http://docs.emulab.net. [Accessed June 2022].

[33] "Cloudlab," [Online]. Available: https://www.cloudlab.us. [Accessed June 2022].

[34] R. Guerraoui and R. Oliveira, "Stubborn Communication Channels," *LSE, D'epartement d'Informatique, Ecole Polytechnique F'ed'erale,* 1996.

[35] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM),* vol. 32, no. 4, pp. 824-840, 1985.

[36] "Github - Implementation of ZeroMQ Go Interface," [Online]. Available: https://github.com/pebbe/zmq4. [Accessed April 2022].

[37] "Visual Studio Code," [Online]. Available: https://code.visualstudio.com/. [Accessed April 2022].

[38] "Ubuntu," [Online]. Available: https://ubuntu.com/. [Accessed April 2022].

[39] "Github - self-stabilizing-mvc," [Online]. Available: https://github.com/acharalampous/self-stabilizing-mvc.

[40] "Go gob package," [Online]. Available: https://pkg.go.dev/encoding/gob. [Accessed May 2022].

[41] "Cloudlab xl170 node type," [Online]. Available: https://www.utah.cloudlab.us/portal/show-nodetype.php?type=xl170. [Accessed June 2022].