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

Constandinos Demetriou

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

May, 2022

APPROVAL PAGE

Master of Science Thesis

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

Presented by

Constandinos Demetriou

Research Supervisor ______________________
Dr. Chryssis Georgiou

Committee Member ______________________
Dr. Ioannis Marcoullis

Committee Member ______________________
Dr. Anna Philippou

University of Cyprus

May, 2022

ABSTRACT

Binary consensus is a problem in which a set of processors must agree on a single binary value. In asynchronous systems, where a subset of the processors may be malicious, this challenge gets more challenging. We study malicious and more serious problems in this work: transient faults. These are temporary violations of the system's operating assumptions that might cause the system's state to change unexpectedly, making recovery impossible without human intervention. We implement an existing protocol for randomized Byzantine-tolerant binary consensus algorithm that is *loosely-self-stabilizing* using the Go programming language and the ZeroMQ communication framework. This approach is optimal in terms of resilience and termination, requires only bounded memory, and ensures that the system will automatically converge to a legal state. Nevertheless, safety violations have a $O(2^{-M})$ likelihood of occurring, where M is a predefined system parameter. This is the first time-free Byzantine-tolerant binary consensus algorithm to make such guarantees, to our knowledge. We describe the first known implementation of this algorithm in this work. With this implementation, we were able to: (a) validate the algorithm; (b) compare the algorithm to its non-stabilizing version and estimate the cost of self-stabilization in terms of processing time and message load; and (c) notice that different failures in the system (whether due to transient faults or malicious behavior) have no significant overhead on recovery and decision time.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my supervisor Dr. Chryssis Georgiou, Professor in the Computer Science Department of University of Cyprus, for his willingness and the opportunity that gave me to work my thesis in a very important problem. He gave me all the support and encouragement throughout the elaboration of my thesis project. Moreover, I would like to give my gratitude to the Postdoctoral Researcher, Dr. Ioannis Marcoullis for providing advice during the meetings and for his excellent support and guidance. This research was supported by the ONISILOS postgraduate scheme of the University of Cyprus, as part of the 2020-2022 project with title: "Towards an Efficient and Self-Stabilizing Byzantine Fault-Tolerant Blockchain Infrastructure". Last but not least, I would like to acknowledge and thank my family for their valuable help during my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This thesis provides an implementation, validation and an experimental analysis of a self-stabilizing Byzantine-tolerant binary consensus algorithm. The solution considers asynchronous message-passing systems under various failure models.

1.1 Motivation

One of the most well-known problem in distributed computing is the *consensus* problem [22], where multiple processors must agree on a common value. When processors try to solve consensus with only two possible decision values, e.g., zero and one, then this task is known as *binary consensus* [29, Ch. 14]. This problem has received a lot of attention recently, because it is considered a fundamental building block of atomic broadcast [29, Ch. 16 and 19]. Since atomic broadcast can be used to implement state machine replication, the functionality is also useful to implement distributed ledgers (e.g., blockchains), due to the fact that many blockchains explicitly solve atomic broadcast to manage block ordering [6, 25].

System failures, let them be hardware failures, power failures, communication interruptions, can have such adverse repercussions on industries, services, and governments, that the commodity of fault-tolerance has major day-today gains. Fault-tolerant systems provide guarantees, given that some assumptions hold. It is possible that in some instances system assumptions are violated. For example, a system might be tolerant to a minority of failures of its processing entities, or to less than a third of those to exhibit malicious behavior.

Byzantine faults [22] represent a type of fault in distributed systems where some processors fail and behave maliciously. Systems that tolerate Byzantine faults work properly, as defined in their specifications, even in the presence of Byzantine faults. If malicious (Byzantine) entities are considered, then only the correct processors are expected to agree this common value.

Fundamental system assumptions are not always guaranteed to hold true over time. Even systems with great availability and dependability might suffer from rare failures. For example, a soft error (some accidental bit-flip) may force a counter to acquire its maximal value, and thus drive the system to either non-progress or to a permanent violation of the system's safety properties. A corrupt program counter or program variable can bring the system to an arbitrary state from which it cannot recover, since it was not anticipated by the system's designers. The system remains useless, requiring human intervention to recover and personnel to be always on-call.

Self-stabilizing systems [15, 14] are designed to automatically recover the system back to its working state and desired behavior. Such systems have a comprehensive approach towards faulty states that usually system designers consider as impossible to reach. In this way, self-stabilizing systems guarantee convergence to a legitimate system state starting from any possible system state, and closure when this legitimate state has been reached, and until the guarantees of the system are violated again.

An overview of the current state of the art, as Chapter 2 shows that there are not enough works related to self-stabilizing byzantine fault tolerant systems. This thesis seeks to provide an implementation and an experimental evaluation of a self-stabilizing binary consensus algorithm.

1.2 Contribution

We present the first, to our best knowledge, implementation and experimental validation and evaluation of a self-stabilizing randomized Byzantine-tolerant algorithm, namely of the algorithm by Georgiou et al. [19]. We use the Go programming language [20] together with the ZeroMQ message-passing library [21]. Also, we perform the experimental validation to make sure of the correctness of our implementation using unit tests. We then proceed to compare this algorithm with an implementation of the original non-stabilizing binary consensus algorithm by Mostefaoui et al. [26]. Moreover, we evaluated the performance overhead which is caused to the presence of Byzantine and transient faults.

1.3 Methodology

Initially, we began by reading several, more general material around Byzantine Fault Tolerance and Self-stabilization. Then, we focused on the Byzantine-tolerant asynchronous binary consensus algorithm with self-stabilizing guarantees by Georgiou et al. [19], which is the algorithm we implemented. Next, we learned and practiced the Go programming language. [20] as well as the ZeroMQ messaging library [33], by developing several small projects in order to get familiar with these two technologies. Also, we studied the implementation of another work [27], to familiarize ourselves with a similar development approach that used the Go programming language with the ZeroMQ framework in order to use the same communication layer. During the implementation, a form of an Agile Software Development process took place. After the implementation and the testing, we ran the algorithm locally on a single machine, for debugging and validating the correct operation of the algorithm. In addition, we evaluated its performance and behavior both locally and in a real-world distributed environment, a cluster of five machines. Finally, we took several execution measurements and compared them with an implementation of the original non-stabilizing binary consensus algorithm by Mostefaoui et al. [26].

1.4 Document Structure

The remaining parts of this thesis are structured as follows: In Chapter 2 we study prior work and related literature about Consensus, Byzantine Fault Tolerant and Self-stabilization. Chapter 3 describes the system's settings and gives a description of the studied algorithm and its functionality. In Chapter 4 we give a thorough explanation of the implementation technologies used in this work: the Go programming language and the ZeroMQ messaging library. Chapter 5 discusses implementation details and important design decisions regarding the tools used. Chapter 6 presents the experiments performed, along with the results and outcomes of the experimental evaluation. We conclude with Chapter 7 where we overview the thesis work, and discuss future research directions of the presented line of work.

Chapter 2

Related Work

We overview related work in the research areas considered by this thesis. In this chapter we present a literature review about the consensus problem, the different type of failures, Byzantine Fault Tolerance and Self-stabilization.

2.1 Consensus Problem

Possibly the most well-known problem in distributed computing is the consensus problem [22] where multiple processors must agree on a common value. In a decentralized system, achieving consensus is one of the most important and most difficult tasks. Many distributed applications, such as cloud computing, service replication, load balancing, and distributed ledgers, e.g., Blockchain, require the system to solve consensus in which all nodes reliably agree on a single value.

The problem has garnered much attention in past and recent times, both on its own, but also because it is considered a fundamental building block of reliable total order broadcast (atomic broadcast) [29]. Since atomic broadcast can be used to implement state machine replication, the functionality is also useful to implement distributed ledgers, e.g., blockchains, and indeed many blockchains explicitly solve atomic broadcast to manage block ordering [6, 25]. This use accounts for the most recent spike

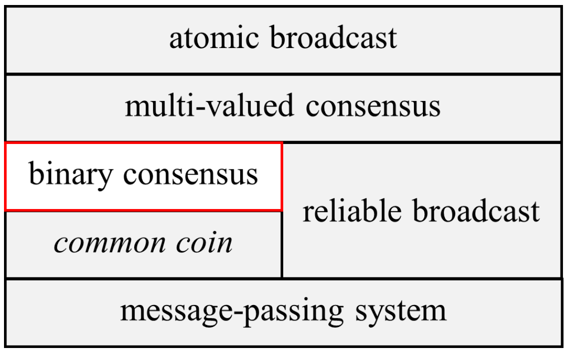


Figure 1: The stack of protocols implementing atomic broadcast. At the current work we are implementing the first Byzantine-tolerant binary consensus module that is loosely-self-stabilizing.

in research interest for the problem. Consensus and atomic broadcast were shown to be equivalent [8]. It is customary to build atomic broadcast on top of a stack of protocols (see Figure 1) including binary and multi-valued consensus [9]. These protocols are designed as successive transformations from one to another. For example, the multi-valued consensus, is implemented on top of a randomized binary consensus and also uses a reliable broadcast protocol. Moreover, the protocols assume that they are built on top of reliable channels, hence bit flips rarely happen.

The problem, as it has been already mentioned, becomes difficult when some nodes may behave arbitrarily or even fail, so the system must be designed in such a way that deals with this inevitability. If faulty entities are considered, then only the correct processors are expected to agree on this common value. Therefore, consensus must satisfy three correctness properties [11]:

1. **Validity**. If a correct processor decides upon a value, then this value was proposed by a correct processor.

2. **Agreement**. If two correct processors decide, then their decided value is identical.

3. **Termination**. All correct processors eventually decide.

2.2 Failures

One of the main challenges of distributed systems is how the system can provide services correctly in the presence of failures. Therefore, a distributed system must be able to deal with various types of failures. The algorithm that we focus on in this work can tolerate Byzantine and arbitrary transient faults. These two types of faults are described in this section.

2.2.1 Byzantine faults

Byzantine faults [22] represent a type of fault in distributed systems where some processors fail and behave maliciously. In contrast to a crash failure, a Byzantine failure does not necessarily mean that the processor stops sending messages. Instead, a Byzantine node can still be active and send malicious messages that do not follow the specification of the algorithm running on the system. The messages sent can simply be corrupted, or they can be carefully constructed by a malicious attacker whose aim is to cause to system to behave incorrectly.

Byzantine faults are permanent, meaning that a Byzantine node acts in a malicious way during the entire lifetime of the system. Compared to crash failures, Byzantine failures are more difficult to handle since it is no longer possible to trust that the messages received follow the specification.

Byzantine faults can occur due to several reasons [7]. As previously mentioned, they can be the result of a malicious attack, where an intelligent attacker compromises one or more nodes in the system and has control over what messages are being sent. However, Byzantine faults do not necessarily happen due to malicious reasons. They can also be the result of hardware and software bugs, where e.g. a buffer overflow in one node causes it to send corrupted or outdated messages.

2.2.2 Transient faults

An arbitrary transient fault represents any possible temporary violation of the assumptions that can happen to a system, except that the algorithm code stays intact [14]. These types of faults include e.g. memory corruption induced by electromagnetic interference and control logic failures at the hardware level. In other words, a transient fault is something that cannot be foreseen or predicted, but rather a fault which violates the nature of the system.

The combination of all things that can go wrong puts the system in an arbitrary state, from which a self-stabilizing algorithm [15] can recover. An algorithm is said to be self-stabilizing if it can recover after the occurrence of transient faults within a bounded number of execution steps, provided that no more transient faults occur during the time of recovery.

2.3 Byzantine Fault-tolerance

The most severe type of failure in distributed computation is the malicious or Byzantine one [22], in which some processors may act arbitrarily by not following the defined algorithm. Systems that tolerate Byzantine faults work properly, as defined in their specifications, even in the presence of Byzantine faults.

2.3.1 Byzantine Consensus

Byzantine Fault Tolerance (BFT) originates from the Byzantine Generals Problem [22]; the hypothetical scenario in which the army of the Byzantine empire has multiple separate divisions surrounding an enemy city, and they have to all agree on a common plan to attack or retreat. All, or at least nearly all, must perform the same action or else they will risk complete failure. The generals can communicate with one another only by messenger. The problem is that some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm [22] to guarantee that all loyal generals decide upon the same plan of action. The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. In-addition, a small number of traitors cannot cause the loyal generals to adopt a bad plan.

In [22], it was proven that the Byzantine Generals Problem and thus consensus, has no solution when at least one-third of the generals are traitors. Hence, the algorithm preserves the properties of optimal resilience, that up to f nodes out of $3f + 1$ in total can be Byzantine. Therefore, the goal of BFT systems is to be able to resist up to one-third of the nodes acting maliciously and continue functioning correctly as long as the other two-thirds of the network reach consensus. For instance, all decentralized blockchains [35] run on consensus protocols that all nodes in the blockchain must follow in order to participate, such as Proof-of-Work and Proof-of-Stake that are Byzantine Fault Tolerant and are thus able to resist up to one-third of the nodes disagreeing.

2.3.2 FLP Impossibility

Strong synchrony distributed systems come with limitations on, for example, availability, since it requires processors to wait for the slowest processors before executing the next step. On the other hand, in asynchronous systems, there is no upper bound on the amount of time processes may take to receive messages.

The landmark FLP impossibility result as proven by Fischer et al. [18], shows that agreement in asynchrony is impossible in the presence of even a single failure. In other words, a distributed asynchronous system cannot reach consensus, in the absence of a mechanism for determining whether or not a processor has crashed or if it is simply taking a long time to respond.

FLP impossibility [18], states that both agreement and termination correctness properties cannot be satisfied in an asynchronous distributed system, if it is to be resilient to at least one fault. Consequently, FLP impossibility states that asynchronous fault tolerant systems cannot simultaneously agreement, validity and termination, thus, fault tolerance and correctness cannot be achieved.

Since then, several ways have been proposed to bypass the impossibilities for the general case [10]. Popular ways to do this, are to introduce some synchrony to the model and assume a known delay to the system's communication. Another way is to employ a failure detection mechanism [2], which is an external mechanism that detects faults. Finally, a different approach is to relax the requirement for a deterministic solution and use randomization [3], for example coin flip algorithms [5].

2.4 Self-stabilization

The self-stabilization paradigm was first introduced by Dijkstra [14]. He calls a system *self-stabilizing* when, "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". A legitimate state can be defined as a system state where the system holds the requirements for the current algorithm. For instance, the requirements for a leader election algorithm are that at one point at most one leader may exist in the system. Legal execution is the way a system should behave, whether it is self-stabilizing or not.

A system is said to be self-stabilizing if and only if, the system is guaranteed to reach its legal execution within a bounded number of execution steps regardless of its initial state [14]. We assume that any transient fault leads to an arbitrary state of the system, possibly with stale information but with the program code intact. Such transient faults could lead to the system having e.g., stale variables, and can potentially affect the outcome of the system, but are not considered to affect the program code. A self-stabilizing system will automatically recover from a transient fault and reach its legal execution after the occurrence of the last transient fault. A recovery period is the period when the system is converging to its legal execution.

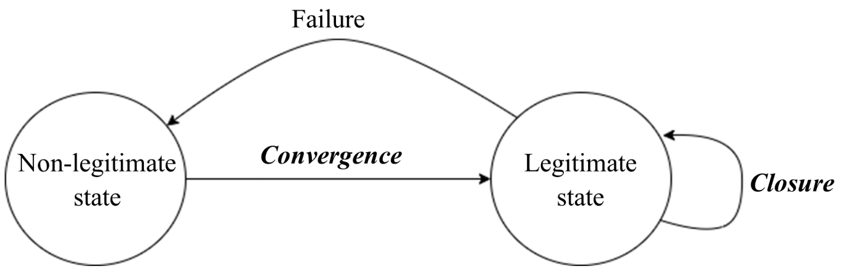


Figure 2: A self-stabilizing system following two properties: convergence and closure

For example, a soft error (some accidental bit-flip) may force a counter to acquire its maximal value, and thus drive the system to either non-progress or to a permanent violation of the system's safety properties. A corrupt program counter or program variable can bring the system to an arbitrary state from which it cannot recover, since it was not anticipated by the system's designers [32]. The system remains useless, requiring human intervention to recover and personnel to always be on-call. Self-stabilizing systems [14, 15, 30] are designed to automatically recover the system back to its working state and desired behavior, from any given state it may end up to after an unanticipated failure.

Self-stabilization was later formally defined by Schneider [30] as: For a system that is self-stabilizing, it satisfies the following two properties, as shown in Figure 2:

1. **Convergence**. Starting from an arbitrary state, convergence property is guaranteed that the system will eventually reach a legitimate state within a finite number of state transitions.

2. **Closure**. Given that the system is in a legitimate state, it is guaranteed to stay in a legitimate state, until the guarantees of the system are violated again.

Furthermore, since transient failures can't always be detected, a process in a self-stabilizing system must keep checking if its local state is legitimate. Meaning if a transient failure occurs, the process will eventually detect that it is no longer in a legitimate state and then take some action. This requires what is referred to as the "do forever loop" or "forever loop".

2.5 Self-stabilizing Byzantine Agreement

The few attempts to tackle the problem of self-stabilizing Byzantine Agreement are very recent.

Daliot and Dolev [12] consider a more severe fault model than permanent Byzantine failures, one in which the system can in addition be subject to severe transient failures that can temporarily throw the system out of its assumption boundaries. Classic Byzantine algorithms cannot guarantee to execute from an arbitrary state, because they are not designed with self-stabilization in mind. They present a self-stabilizing Byzantine agreement algorithm that reaches agreement among the correct nodes in optimal time, by using only the assumption of bounded message transmission delay. In the process of solving the problem, two additional important and challenging building blocks were developed: a unique self-stabilizing protocol for assigning consistent relative times to protocol initialization and a Reliable Broadcast primitive that progresses at the speed of actual message delivery time. The FLP impossibility results is bypassed deterministically with timing assumptions, which is a type of synchronization.

Dolev et al. [16] contribute the first self-stabilizing State Machine Replication (SMR) service that is based on failure detectors without use of clock synchronization and timeouts. They suggest an implementable self-stabilizing failure detector to monitor both responsiveness and the replication progress. They thus encapsulate weaker synchronization guarantees than the previous self-stabilizing BFT SMR solution. They follow the seminal paper by Castro and Liskov [7] of Practical Byzantine Fault Tolerance and focus on the self-stabilizing perspective. This work can aid towards building distributed blockchain system infrastructure enhanced with the self-stabilization design criteria.

Ongoing work by Lundström et al. [23, 24] provides self-stabilizing binary consensus and multivalued consensus (with the "indulgence" and "zero-degradation" characteristics). These algorithms can form the basis for asynchronous randomized SMR, but they are not Byzantine tolerant.

Georgiou et al. [19] present the first loosely-self-stabilizing fault-tolerant asynchronous solution to binary consensus in Byzantine message-passing systems. Binary consensus is the agreement where the set of values that can be proposed is either zero or one and it is a fundamental building block for other "flavors" of consensus, e.g., multivalued, or vector, and of total order broadcast. This is achieved via an instructive transformation of MMR [26] to a self-stabilizing solution that can violate safety requirements with the probability $Pr = O(2^{-M})$, where $M \in Z+$ is a predefined constant. The obtained self-stabilizing version of the MMR algorithm considers a far broader fault-model since it recovers from transient faults. Additionally, the algorithm preserves the MMR's properties of optimal resilience and termination, i.e., $t < n/3$, and $O(1)$ expected decision time. Furthermore, it only requires a bounded amount of memory. FLP Impossibility results is bypassed non-deterministic with randomization.

Finally, Duvignau et al. [17] focus on a fundamental module for dependable distributed systems: a self-stabilizing Byzantine tolerant algorithm for multivalued consensus for asynchronous message-passing systems. Multivalued consensus assumes that each non-faulty process advocates for a single value from a given set V. In addition to tolerating Byzantine and communication failures, self-stabilizing systems can automatically recover after the occurrence of arbitrary transient faults. These faults represent any violation of the assumptions according to which the system was designed to operate (provided that the algorithm code remains intact).

Chapter 3

The Algorithm

In this chapter we present the studied loosely-self-stabilizing randomized Byzantine-tolerant binary consensus algorithm by Georgiou et al. [19]. This is the algorithm we implemented, validated and evaluated.

3.1 System Settings

The system considered is asynchronous and message-passing, comprising n processors $p_1, p_2, \ldots, p_n$ each having its own unique identity. Each pair of processors communicates via a bounded-capacity bidirectional channel. No assumptions are made on the communication delays. Channels are private, i.e., the adversary cannot read or change the contents of a message, but due to the channels' boundedness, messages may be dropped if the channel is full. For liveness, we assume that a message sent infinitely often will be received infinitely often [15].

Failure model. Processors may exhibit Byzantine (malicious) behavior by not following the algorithm specifications. Because of a well-known impossibility [22], the algorithm can only guarantee safety if the number of Byzantine processors t is less than a third of the total number processors' set, i.e., $t <$

$n/3$. Such processors may send malicious messages to each other and collaborate in their operations, but they cannot impersonate another processor.

An arbitrary transient fault represents any possible *temporary* violation of the assumptions that can happen to a system, except that the algorithm code stays intact [14]. For example, a soft error (some accidental bit-flip) may force a counter to acquire its maximal value, and thus drive the system to either non-progress or to a permanent violation of the system's safety properties. Also, corruption of the system state can bring the system to an arbitrary state. This can lead to a violation of safety.

Random bits. The algorithm of [19] that we implement, uses a common coin to provide the same random bit to each processor at every round. They assume the existence of a self-stabilizing random bit algorithm such as the one by Ben-Or et al. [4]. For this work, we consider the self-stabilizing common coin construction as a black box. In Section 5.1 we provide further details about how we implemented its functionality.

3.2 Loosely-self-stabilizing Systems

The occurrence of these arbitrarily transient faults can cause unpredictable changes in the system state. Dijkstra [14] considers that these violations drive the system to an arbitrary state from which a self-stabilizing system should recover when modeling the system. Dijkstra specifies that the system must recover after the last transient-fault occurrence, and that once it has recovered, it must never violate the task specification.

There are currently no known approaches to meet Dijkstra's self-stabilizing design criteria in the context of the studied problem and fault model. Loosely-self-stabilizing systems [31] need that once the system has recovered, it can only violate the safety specifications on rare and brief occasions. Although it is a weaker design criterion than Dijkstra's, the occurrence of violations can be made to be extremely rare.

The algorithm we present is loosely-self-stabilizing [19] and it considered guarantees convergence after transient faults, but safety may be violated after recovery with a probability $O(2^{-M})$, where M is a positive integer constant that is predefined to the system. This is a weaker property, compared to Dijkstra's definition of self-stabilization that guarantees closure, i.e., no violations once convergence has taken place.

3.3 Algorithm Description

We present a more verbal version of the algorithm to make the reading more understandable. The reader can check for more technical details of the algorithm in [19].

Proposal and initialization. The algorithm is initiated by a call to $propose(v)$ from an upper layer module (e.g., multi-valued consensus as per Figure 1), where v is the proposed initial value (line 3). The round counter r, and the two structures holding the estimated values: $est[M+1][n]$ and $aux[M+1][n]$ is part of the local state. The fields $est_i[r'][j]$ and $aux_i[r'][j]$ hold the corresponding value that p_j reported to p_i on p_j's round r'. Field $est_i[M+1][i]$ is expected to hold p_i's final decided value.

The local variables are then initiated to some default values. This initialization is appropriate in the typical case where a transient fault does not occur, but a transient fault may apparently contribute arbitrary values to the structures. This gives the rest of the algorithm a boost, allowing it to recover to the expected system state.

Self-stabilizing algorithms run within a do-forever loop, which means that they theoretically run in an infinite execution. If there is an initial state, then the algorithm proceeds to the consensus part in the next lines (line 15). The algorithm starts by increasing the round counter, unless this has reached the upper bound of $M+1$ (line 16).

The consensus procedure is performed by entering the repeat-until loop. To discover possible corruptions related to transient faults, the program first runs various consistency tests. Line 18 checks whether the initial estimate $est[0][i]$ is not a binary value (indicating a state corruption) and if this holds, the value is corrected to one of the binary pair. The second check if the est and aux values of P_i in rounds before the current round r are absent (line 19).

Phase 1: Query for the estimated values. Processor p_i informs other processors of its own value (line 21) and receives the other processors' values through the communication mechanism of lines 25–27. The algorithm is wait-free since it transmits messages continually and continues on once the conditions are met.

Phase 2: Informing about results of Phase 1. Once a value x is reported by $2t + 1$ processors for round r and if for the respective round the $aux[r][i]$ field is either null or is not the value already held by p_i, then x is added to $aux[r][i]$ (line 20). Once $aux[r][i]$ has a non-null value, the round enters this second phase. Moreover, the 2t+1 requirement assures that this set contains at least t+1 correct processors, which make up the majority of the correct ones.

Phase 3: Attempt to decide on a single binary value. The $infoResult()$ macro returns a non-empty set then the algorithm attempts to decide (line 23), calling function $tryToDecide()$. If no unique value exists, then the value added to $est[r][i]$ is the value of the random bit. If a unique value exists, it is used as the round r value. If this is also the random bit for round r, then this is the value that will be used in this case. In this case the $decided()$ function is called and the decision is written on all the fields concerning p_i between r and $M + 1$.

The receiver's side of the communication (lines 25–27) stores the received est and aux values about p_j's round rJ values and sends the values of p_i for the corresponding round. A flag breaks the vicious cycle of constantly sending these messages back and forth.

Finally, recovering the result is provided to the upper layer by an interface named $result()$ which returns the value of $est[M + 1][i]$ if this has been decided, or $\perp$ if no decision has been made. If the module is unable to make a decision due to an error, the interface returns Ψ.

As an optimization, line 24 allows for a fast decision once at least $t + 1$ processors have decided. This is because, among the $t+1$ processors' set there must be at least one correct processor. If a correct processor has made a decision, subsequent correct processors may make a decision based on this value rather than waiting for support with a higher threshold at line 21.

Algorithm 1: Loosely-self-stabilizing Byzantine-tolerant binary consensus that uses M iterations and violates safety with a probability that is in $\mathcal{O}(1/2^M)$; code for p_i. (Simplified version of the one in [19].)

```
variables: $r \in$ is the round number counter; Structure $est[M+1][n]$ holds every processor's
  reported value per round for the first phase. $est[r][j]$ holds $p_j$'s value for round $r$; Structure
  $aux[M+1][n]$ holds the values per round for each processor for the information phase.
  $aux[r][j]$ holds $p_j$'s value for round $r$;
constants: $initState$ is the default local state of the system with $r = 0$ and empty estimate and
  auxiliary value structures; $M$ is a system defined positive integer;
operations: propose$(v_p)$ do $\{(r, est, aux) \leftarrow initState$; Set $est$ for $p_i$'s round 0 to $\{v_p\}\}$;
result() do {if $(est[M{+}1][i] = \{v\})$ then return $v$ else if $(r \geq M \wedge$ infoResult() $\neq \emptyset)$ then
  return $\Psi$ else return $\bot$;}
macros: $binValues(r, x)$ return the values held in $est[r][\bullet]$ by at least $x$ processors for round $r$
infoResult() do {if there exists a set of more than $n - t$ processors for which value $val$ in
  $aux_i[r][]$ belongs to $binValues(r, 2t{+}1)))$ then return $val$ else return $\emptyset$;}
functions: decide$(x)$ begin
    foreach round $r' \in \{r, \ldots, M{+}1\}$ do
        if $(est[r'][i] = \emptyset \vee aux[r'][i] = \bot)$ then $(est[r'][i], aux[r'][i]) \leftarrow (\{x\}, x)$;
    $r \leftarrow M{+}1$;
tryToDecide$(values)$ begin
    if $(values \neq \{v\})$ then $est[r][i] \leftarrow \{\mathbf{randomBit}(r)\}$;
    else $\{est[r][i] \leftarrow \{v\}$; if $(v = \mathbf{randomBit}(r))$ then decide$(v)\}$;
do forever begin
    if $((r, est, aux) \neq initState)$ then
        $r \leftarrow \min\{r{+}1, M{+}1\}$;
        repeat
            if The initial estimate $est[0][i]$ is not a unique binary value then Reset the value to
              any value;
            foreach round $r' \in \{1, \ldots, r{-}1\}$ where $est[r'][i]$ or $aux[r'][i]$ are empty/null do
              Reset these values to $est[i][0]$ ;
            if $\exists$ value $v \in binValues(r, 2t+1)$ that is not already in the $aux[r][i]$ then
              $aux[r][i]$;
            foreach $p_j \in \mathcal{P}$ do send $\mathrm{EST}(r, est[r{-}1][i] \cup binValues(r, t{+}1), aux[r][i])$ to $p_j$
        until infoResult() $\neq \emptyset$;
        tryToDecide(infoResult());
        if $t + 1$ processors have decided on value $w$ then decide$(w)$ ;
upon receipt of EST message from $p_j$ for round $rJ$ begin
    store values in $est[rJ][j]$ and $aux[rJ][j]$;
    return EST message with $p_i$'s values in round $rJ$, as in $est[rJ-1][i]$ and $aux[rJ][i]$ ;
```

Chapter 4

Technological Background

The self-stabilizing randomized Byzantine tolerant binary consensus algorithm was implemented in the Go programming language [20], embedded with the ZeroMQ messaging library [33, 34]. This chapter provides a detailed description of these technologies.

4.1 Go Programming Language

Go is a statically-typed, compiled programming language with incredibly fast compilation speeds designed at Google [20]. Some of its core developers were members of the UNIX team at Bell Labs like Ken Thompson and Rob Pike and were primarily motivated by their shared dislike of C++. Primary goals were to have a programming language that has static typing and run-time efficiency, that is readable and usable similar to Python and Javascript, trying to unify programming languages developers use within Google, and to have high-performance networking and multiprocessing capabilities.

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

Concurrency in Go. The Go language has built-in facilities, as well as library support, for writing concurrent programs. Mainly, it deploys concurrency following the CSP paradigm, which is a formal language for describing patterns of interaction in concurrent systems. It does so by providing goroutines, channels, and a rich standard library package featuring most of the classical concurrency control structures. Firstly, goroutines are light-weight coroutines, or more accurately "green-threads", which are initiated with a function call prefixed with the "go" keyword, and so that function starts in a new concurrency "thread". As shown in the Listing 1 in line 5, $go\ hello()$ starts a new goroutine. Now the $hello()$ function will run concurrently along with the $main()$ function. The main function runs in its own goroutine and it's called the main goroutine. Channels provide the ability to send messages between goroutines, which are stored in a FIFO order that allows goroutines to wait either when they try to pull a message from an empty channel or when they try to push a message to a full channel. Therefore to avoid blocking on a full channel, the built-in "select" statement can be used to implement non-blocking communication on multiple channels. In the Listing 2, we create a $done$ bool channel in line 6 and pass it as a parameter to the $hello$ goroutine. In line 8 we are receiving data from the done channel. This line of code is blocking which means that until some goroutine writes data to the done channel, the control will not move to the next line of code. The line of $code < -done$ receives data from the $done$ channel but does not use or store that data in any variable. Now we have our main goroutine blocked waiting for data on the $done$ channel. The $hello$ goroutine receives this channel as a parameter, prints "Hello world goroutine" and then writes to the $done$ channel. When this write is complete, the main goroutine receives the data from the done channel, it is unblocked and then the text main function is printed. Thus, the Go programming language makes a perfect fit for concurrent processing and networked systems.

```
func hello() {
    fmt.Println("Hello world goroutine")
}

func main() {
    go hello()
    fmt.Println("main function")
}
```

Listing 1: A simple example that shows how to create goroutines in Go programming language

```
func hello(done chan bool) {
    fmt.Println("Hello world goroutine")
    done <- true
}

func main() {
    done := make(chan bool)
    go hello(done)
    <-done
    fmt.Println("main function")
}
```

Listing 2: A simple example that shows how to use channels in Go programming language

4.2 ZeroMQ Messaging Library

ZeroMQ is a high-performance asynchronous messaging library [33], designed to be used in distributed and concurrent applications. ZeroMQ provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. The library's API is designed to resemble Berkeley sockets, and although it looks like an embeddable networking library, in reality it acts like a concurrency framework. Originally the zero in ZeroMQ was meant as "zero broker" or "zero latency", however since then, it has come to encompass different goals like zero cost or zero waste, and generally, "zero" refers to the culture of minimalism that permeates the ZeroMQ project. Among its many benefits, ZeroMQ has sockets that carry atomic messages across various transport protocols like in-process, inter-process, TCP, or multicast, it has a score of language APIs and also runs on most operating systems. Furthermore, ZeroMQ is fast enough to be the fabric for any clustered product and its asynchronous I/O model can support scalable multicore applications, built for asynchronous message processing tasks.

Socket Types. One of the most important advantages of ZeroMQ is that it provides a range of sockets which generalize the traditional IP and Unix domain sockets, each of which can be combined and form N-to-N messaging patterns. Sockets provided by ZeroMQ are [34]:

- *REQ*: Sockets used by a client to send requests to and receive replies from a service. REQ sockets must follow the pattern send, receive, send, receive.
- *REP*: Sockets used by a service to receive requests from and send replies to a client. REP sockets must follow the pattern receive, send, receive, send.
- *DEALER*: Talks to a set of anonymous peers, sending and receiving messages using round-robin algorithms. Works as an asynchronous replacement for REQ, for clients that talk to REP or ROUTER servers.
- *ROUTER*: Talks to a set of peers, using explicit addressing so that each outgoing message is sent to a specific peer connection. Works as an asynchronous replacement for REP, and is often used for servers that talk to DEALER clients.
- *PUB*: Used by a publisher to distribute data. Messages sent are distributed to all connected peers. This socket type is not able to receive any messages.

- *SUB*: Used by a subscriber to subscribe to data distributed by a publisher. Initially a SUB socket is not subscribed to any messages. The send function is not implemented for this socket type.
- *XPUB*: Same as PUB except that you can receive subscriptions from the peers in the form of incoming messages.
- *XSUB*: Same as SUB except that you subscribe by sending subscription messages to the socket.
- *PUSH*: Talks to a set of anonymous PULL peers, sending messages using a round-robin algorithm. It has no receive operation.
- *PULL*: Talks to a set of anonymous PUSH peers, receiving messages using a fair-queuing algorithm.
- *PAIR*: Socket that can only be connected to a single peer at any one time. No message routing or filtering is performed on messages sent over a PAIR socket.
- *CLIENT*: Talks to one or more SERVER peers. If connected to multiple peers, it scatters sent messages among these peers in a round-robin fashion, and it does not drop messages in normal cases.
- *SERVER*: Talks to zero or more CLIENT peers. Each outgoing message is sent to a specific peer. A SERVER socket can only reply to an incoming message.

Messaging Patterns. Using and combining these socket types, various messaging patterns or architectures can be built depending on the topology needed. ZeroMQ patterns are implemented by pairs of sockets with matching types. The core built-in messaging patterns [34] ZeroMQ offers are:

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

Listing 3 shows a server that creates a socket of type response (REP), binds it to port 5555 and then waits for messages. Also, in this example we have zero configuration, we are just sending strings. Listing 4 illustrates a client that creates a socket of type request (REQ), connects and starts sending messages. Both the send and receive methods are blocking (by default). For the receive it is simple: if there are no messages the method will block. For sending it is more complicated and depends on the socket type. For request sockets, if the high watermark is reached or no peer is connected the method will block. The DEALER-ROUTER sockets have similar functionality to the REQ-REP sockets with the difference that the DEALER-ROUTER sockets allow asynchronous communication without blocking.

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

There are more ZeroMQ patterns that are still in draft state:

- *Client-Server*: Used for allowing a single server to talk to one or more clients. The client always starts the conversation, after which either peer can send messages asynchronously, to the other.

```go
func main() {
    zctx, _ := zmq.NewContext()
    s, _ := zctx.NewSocket(zmq.REP)
    s.Bind("tcp://*:5555")

    for {
        // Wait for next request from client
        msg, _ := s.Recv(0)
        log.Printf("Received %s\n", msg)

        // Sleep
        time.Sleep(time.Second * 1)

        // Send reply back to client
        s.Send("World", 0)
    }
}
```

Listing 3: Implementation example for a server in Go programming language using REP sockets that provided from the ZeroMQ message library. The server receives a request from the client and replies with a string.

```go
func main() {
    zctx, _ := zmq.NewContext()
    // Socket to talk to server
    fmt.Printf("Connecting to the server...\n")
    s, _ := zctx.NewSocket(zmq.REQ)
    s.Connect("tcp://localhost:5555")

    // Do 10 requests, waiting each time for a response
    for i := 0; i < 10; i++ {
        fmt.Printf("Sending request %d...\n", i)
        s.Send("Hello", 0)
        msg, _ := s.Recv(0)
        fmt.Printf("Received reply %d [ %s ]\n", i, msg)
    }
}
```

Listing 4: Implementation example for a clinet in Go programming language using REQ sockets that provided from the ZeroMQ message library. The client sends a string to server and receives a reply.

- *Radio-Dish*: Used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion.

As we explain in Section 5.2 for the purpose of our work we have used REQ/REP (request/reply) pair of sockets to communicate with other processors. Our choice led us to the challenge of establishing asynchronous communication with synchronous REQ/REP sockets.

Chapter 5

Implementation

In this chapter, we describe the system implementation. We begin with a short mention of implementation methodology and decisions. We then proceed to a more concrete discussion regarding the communication layer and the structure of the project.

5.1 Implementation Methodology and Decisions

As previously mentioned, for the implementation of the self-stabilizing randomized Byzantine tolerant algorithm for binary consensus we used Go as the programming language, embedded with the ZeroMQ messaging library. The implementation is open for everyone on GitHub [13].

Programming language – Go. In this work we decided to use the Go language because it is suggested to provide functionality for implementing efficient applications with scalable concurrency mechanisms known as goroutines and channels. As already mentioned on the Section 4.1, goroutines are lightweight coroutines, or more accurately "green" threads associated with less overhead and the Go runtime is very efficient in the handling of these goroutines. Also, channels provide the ability to send messages between goroutines. Since we needed to have each processor running concurrently, goroutines greatly facilitated the need for multi-threading support, and channels are the means to communicate and synchronize with concurrent goroutines [20]. Furthermore, our results will be compared with the results of the non-self-stabling algorithm which were developed also using Go [27], thus having them all implemented in the same programming language makes the comparison and evaluation of the algorithms more valid and accurate. Finally, even though Go is a newly developed programming language, due to the fact that it is delivered by Google and is really hyped in the market, its documentation on the Internet is really huge and remarkably helped us during the implementation process.

Communication implementation – ZeroMQ. When catering for the inter-processor communication, we tried to employ a tool that would enable us to tightly map the assumptions of the system model (cf. Section 3.1). This lead us to use ZeroMQ as our messaging library because it is a high-performance asynchronous messaging library that provides a variety of sockets that can be used to deploy a vast amount of distributed messaging patterns in any networked topology required. As previously stated in Section 4.2, ZeroMQ is state-of-the-art in terms of speed, and reliability and by using hidden message queues makes the delivery of messages guaranteed. Moreover, it has a variety of programming language APIs and runs on most operating systems. Another reason we used the messaging framework ZeroMQ is to have the same communication layer with the implementation of the non-self-stabilizing algorithm [27] to have a fair comparison in our results. Section 5.2 provides details about the ZeroMQ messaging patterns we used.

Common coin. In our work, we implemented a random bit function that simulates a common coin using the rand package from the math library [28] (see Listing 5). Specifically, when a processor calls the random bit function, the rand package is initialized with a seed equal to the *round id* and then the random bit is returned. In this way the random bit function returns the same random bit to all processors on the same round. Therefore, we can visualize the common coin as a black block. Although some self-stabilizing communication protocols exist, there is none to our knowledge complete self-stabilizing random bit implementation.

```
func randomBit(round_id int) int {
    // Initialized with a seed equal to the round id
     rand.Seed(int64(round_id))

     // Generate a random bit (0 or 1)
     random_number := rand.Intn(2)

     return random_number
}
```

Listing 5: Implementation of the random bit function that simulates a common coin

Note. The goal of this project was to validate and evaluate the algorithm. We opted to restrict the use of self-stabilizing components to the binary consensus module itself in this vein. This allowed us to analyze execution time and other issues of self-stabilization that had to do only with this protocol. This was especially true when comparing this algorithm with a non-self-stabilizing one. Therefore, in order to have a fair comparison of the experimental results of the two algorithms, it is important that both implementations are based on the same programming language (Go), have the same communication framework (ZeroMQ) and have the same common coin (random bit device as black box).

5.2 Communication

For the implementation of the communication layer, we used the messaging framework ZeroMQ as already mentioned. Specifically, each processor has a REQ/REP (request/reply) pair of sockets to communicate with other processors, as shown in Figure 3. REQ sockets used by a client to send requests to and receive replies from a service and REP sockets used by a service to receive requests from and send replies to a client.

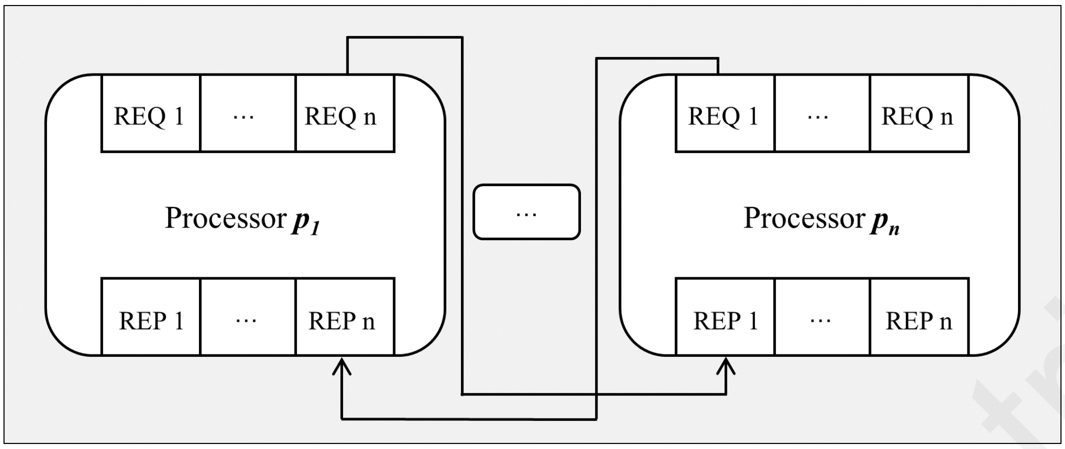


Figure 3: Communication architecture with ZeroMQ sockets

The main reason we picked the REQ/REP pair of sockets over DEALER/ROUTER is that they were used in the implementation of the non-self-stabilizing algorithm [27], and as previously stated we would like to have a common communication layer to compare the experimental findings fairly. Moreover, REQ/REP sockets are the most reliable ones and and their development in the Go programming language is quite simple.

Our decision to use REQ/REP sockets led us to a significant challenge: to establish asynchronous communication with synchronous REQ/REP sockets. We had to avoid the case where a processor might block waiting for another processor's response. Thus, we "emulated" asynchrony using Go, by having a form of a timeout and retransmission of messages, in order to match the asynchrony demands of the system settings. In particular, we used Go goroutines, channels and the `select` statement alongside the `timeticker` functionality that Go offers.

In this way processors intercommunicate with the use of their corresponding pair, but now a processor that will receive a request, will immediately reply with an empty message, rendering its REP socket available for the next request. With this solution, there will not be cases where REP sockets try to receive twice consecutively, or REQ sockets trying to send twice, something that would otherwise cause a system crash. Listing 6, Listing 7 and Listing 8 illustrate a part of the implementation of the communication process.

```go
func InitializeMessenger() {
    for i := 0; i < variables.N; i++ {

        if i == variables.ID {
            continue // Not myself
        }
        ...
        // SendSockets initialization to send information to other
        // servers
        SendSockets[i], err = Context.NewSocket(zmq4.REQ)
        if err != nil {
            logger.ErrLogger.Fatal(err)
        }

        // ReceiveSockets initialization to get information from other
        // servers
        ReceiveSockets[i], err = Context.NewSocket(zmq4.REP)
        if err != nil {
            logger.ErrLogger.Fatal(err)
        }
    }
}
```

Listing 6: Initialization of the send/receive sockets

```go
func TransmitMessages() {
    // Send the message to all processors except myself
    for i := 0; i < variables.N; i++ {
        if i == variables.ID {
            continue // Not myself
        }

        // Initializes them with a goroutine and waits forever
        go func(i int) {
            // For each message in channel
            for message := range MessageChannel[i] {

                // Send the message
                _, err = SendSockets[i].SendBytes(w.Bytes(), 0)
                if err != nil {
                    logger.ErrLogger.Fatal(err)
                }

                // Receive the answer
                _, err = SendSockets[i].Recv(0)
                if err != nil {
                    logger.ErrLogger.Fatal(err)
                }
            }
        }(i)
    }
}
```

Listing 7: Implementation of the function that sends the messages to the other processors

```go
func Subscribe() {
	// Gets messages from other servers and handles them
	for i := 0; i < variables.N; i++ {
		if i == variables.ID {
			continue // Not myself
		}

		// Initializes them with a goroutine and waits forever
		go func(i int) {
			for {
				message, err := ReceiveSockets[i].RecvBytes(0)
				if err != nil {
					logger.ErrLogger.Fatal(err)
				}

				go HandleMessage(message)

				_, err = ReceiveSockets[i].Send("", 0)
				if err != nil {
					logger.ErrLogger.Fatal(err)
				}
			}
		}(i)
	}
}
```

Listing 8: Implementation of the function that receives the messages from the other processors

5.3 Project Structure

The implementation includes a Go project which consists of several packages:

```
self-stabilizing-binary-consensus
├─ modules
│  ├─ self-stabilizing-binary-consensus.go
│  └─ non-self-stabilizing-binary-consensus.go
├─ messenger
│  └─ messenger.go
├─ config
│  ├─ ip.go
│  ├─ local.go
│  └─ scenario.go
├─ types
│  ├─ message.go
│  ├─ bc_message.go
│  └─ ssbc_message.go
├─ variables
│  └─ variables.go
├─ logger
│  └─ logger.go
├─ threshenc
│  ├─ key_generator.go
│  ├─ key_reader.go
│  └─ sign_and_verify.go
└─ main.go
```

The implementation of the self-stabilizing binary consensus algorithm is located in the package *modules*. In the *config* package, the code for configuring the IP addresses of the system and configuring the execution scenario is located. Additionally, a *logger* is implemented that writes info and errors in text files for the monitoring of the system's operation. The *types* package contains all the necessary Go structs and type aliases required for the messages that are exchanged among the different processors. Also, the *variables* package contains the main variables and constants that are shared between modules, such as the number of processors and Byzantine nodes. Finally, *messenger* is the package responsible for sending and receiving messages between clients and servers and among the processes as well.

Modules. In the modules package as already mentioned, we have implemented the self-stabilizing randomized Byzantine tolerant binary consensus algorithm by Chryssis et al. [19] and the non-self-stabilizing randomized Byzantine tolerant binary consensus by Mostefaoui et al. [26]. The binary consensus protocol is a part of a stack of protocols (see Figure 1) which at the top build the atomic broad. If all these protocols become self-stabilizing then we will be able to have a self-stabilizing atomic broadcast and thus it will be easier to create self-stabilizing blockchains in the future.

The non-self-stabilizing binary consensus protocol [9], consists basically of two (2) algorithms, BC and BVB, with BC being the main one that executes the consensus procedure and BVB being the algorithm that broadcasts the messages and fills the bin_values set. In contrast, in self-stabilizing binary consensus protocol [19], the BC and BVB algorithms merge into a single algorithm.

Messenger. In the messenger package, the initialization of the sockets takes place as well as some basic functions that send the messages to other processors and receive messages. Except for these, we also have a couple of functions that in case we are in a scenario that not all servers act non-faulty, the messages sent by Byzantine nodes are modified before transmission to try and harm consensus among the processes.

Config. For local development, the mapping of network addresses to nodes happens in the *local.go* file. It contains four maps of integers to strings, each map storing the corresponding address of the appropriate socket type, and we play with the localhost and different ports representing each processor or client, as illustrated in Listing 9. Similarly, for the real-world configuration, instead of tinkering with ports, each processor has its own computer and IP address found in the *ip.go* file. Table 1 shows the IP addresses of the cluster machines used in the experimental evaluation. Finally, *scenario.go* is responsible for configuring the scenario of execution the servers run, which is basically a flag that allows us to use it in our code and determine how a process needs to act. Finally, *scenario.go* is responsible for configuring the scenario of execution the servers run, which is basically a flag that allows us to use it in our code and determine how a process needs to act.

```go
func InitializeLocal(){
    RepAddresses = make(map[int]string, variables.N)
    ReqAddresses = make(map[int]string, variables.N)
    ServerAddresses = make(map[int]string, variables.Clients)
    ResponseAddresses = make(map[int]string, variables.Clients)
    for i := 0; i < variables.N; i++ {
        RepAddresses[i] = "tcp://*:"+
            strconv.Itoa(4000+(variables.ID*100)+i)
        ReqAddresses[i] = "tcp://localhost:"+
            strconv.Itoa(4000+(i*100)+variables.ID)
    }
    for i := 0; i < variables.Clients; i++ {
        ServerAddresses[i] = "tcp://*:"+
            strconv.Itoa(7000+(variables.ID*100)+i)
        ResponseAddresses[i] = "tcp://*:"+
            strconv.Itoa(10000+(variables.ID*100)+i)
    }
}
```

Listing 9: IP configuration for local execution

Types. Since most of the messages that are exchanged in the system are complex structures containing multiple fields, and ZeroMQ sends messages as a sequence of bytes, a method of serialization and deserialization of structs is necessary, so for this requirement we used Gob [1]. Gob is a package included in Go's standard library, and it manages streams of bytes exchanged between a transmitter and a receiver. Encoding with Gob returns an array of bytes while decoding with Gob builds a struct

Machine ID	ID Address
0	10.16.12.56
1	10.16.12.11
5	10.16.12.100
6	10.16.12.212
8	10.16.12.105

Table 1: IP addresses for cluster machines

from an array of bytes. Gob supports encoding and decoding of all Go's built-in types, but to be able to encode/decode a complex struct, that struct has to implement the GobEncoder/GobDecoder interface by implementing the two (2) basic methods.

So here comes the types package, in which we implemented all the basic types of messages we need for our implementation, like SSBCMessage (see Listing 10). The message.go is the general type in which processors' messages are wrapped. Its structure is composed of the fields Payload, which is an array of bytes, a string Type that denotes the type of the payload in terms of the name of the structure, and an integer From denoting the identity of the sender processor. When a processor receives an incoming message, it decodes received bytes into a Message struct, and then a switch case is applied on the structure's Type field. The message's payload is decoded as well to the appropriate struct type, and the resulting struct is consequently written to the corresponding channel, in which the consumer module will read from. This pattern succeeds in hiding delays of sending and receiving messages, making the message exchange feel more organic.

Variables. The variables package just consists of a file that contains all the vital variables and constants that are used generally in the project. Some of them are, the processor's ID, the number of processors, Byzantine nodes in the system and whether the execution is local or in real-world.

```go
// SSBCMessage - Self-Stabilizing Binary Consensus EST message struct
type SSBCMessage struct {
    Identifier int
    Flag       bool // aJ
    Round      int  // rJ
    Est_0      int  // vJ[0]
    Est_1      int  // vJ[1]
    Aux_0      int  // uJ[0]
    Aux_1      int  // uJ[1]
}
// GobEncode - Binary Consensus message encoder
func (estm SSBCMessage) GobEncode() ([]byte, error) {
    w := new(bytes.Buffer)
    encoder := gob.NewEncoder(w)
    err := encoder.Encode(estm.Identifier)
    if err != nil {
        logger.ErrLogger.Fatal(err)
    }
    ...
    return w.Bytes(), nil
}
// GobDecode - Binary Consensus message decoder
func (estm *SSBCMessage) GobDecode(buf []byte) error {
    r := bytes.NewBuffer(buf)
    decoder := gob.NewDecoder(r)
    err := decoder.Decode(&estm.Identifier)
    if err != nil {
        logger.ErrLogger.Fatal(err)
    }
    ...
    return nil
}
```

Listing 10: Example of SSBCMessage type of message with GobEncode and GobDecode

Threshenc. Moving now to the threshenc package, it is what is called Trusted Dealer. Basically, in this package using built-in libraries of Go, Verification and Secret key-pairs (or Public and Private), for each one of the processors are being created before the processors start their execution, and these key-pairs are stored locally. Then, when each processor starts its execution it reads and keeps the Verification keys of all the processors but only its own Secret key, that is why it is called secret. Therefore, the overhead that the keys have on each processor is just the time it takes to read them from the local files and not their generation too, as that procedure takes place before the execution.

The reason why these key pairs are needed even though the implemented algorithm does not use any digital signatures or threshold encryption schemes, is simply to have a method of validation for each message that is received, to guarantee that the sender is valid. Therefore, using the *sign_and_verify.go* file (see Listing 11), each processor signs the message before sending using their Secret key, appends the signature to the message and then sends it. When the message is delivered, the receiver process verifies that the signature is valid using the Verification key of the sender. If the verification procedure of the message is correct, then the message is consumed from the algorithm, but if it is not correct then it means it is a malicious one and thus it is dropped immediately.

```
func SignMessage(message []byte) []byte {
    hash := sha256.New()
    _, err := hash.Write(message)
    hashSum := hash.Sum(nil)
    signature, err := rsa.SignPSS(rand.Reader, SecretKey, crypto.SHA256,
        hashSum, nil)
    return signature
}

func VerifyMessage(message []byte, signature []byte, i int) bool {
    hash := sha256.New()
    _, err := hash.Write(message)
    err = rsa.VerifyPSS(VerificationKeys[i], crypto.SHA256, hash.Sum(nil),
        signature, nil)
    if err != nil {
        return false
    }
    return true
}
```

Listing 11: Sign and Verify methods

Chapter 6

Experimental Validation and Evaluation

The first key part of our work was to validate the correctness of the algorithm. This was achieved by emulating different failures and scenarios (by injecting faults by code), and based on the results validating the functionality of the algorithm. The second key part consisted of performing a preliminary evaluation, with respect to the performance of the implementation.

6.1 Experimental Scenarios

A distributed algorithm such as the one implemented must be able to deal with various types of failures. Validating our implementation involved proving tolerance against failures. To this end, we constructed scenarios where either processors were mimicking malicious behaviors (detailed is Section 6.1.1), or the system state was changed to imitate the results of transient fault-induced corruptions (documented in Section 6.1.2).

6.1.1 Byzantine attacks

Normal Scenario. This is the failure-free scenario where no Byzantine behavior exists. Byzantine processors act as correct. This allows us to take the baseline measurements for our comparison with next scenarios where Byzantine behavior is expressed in several forms.

Idle Attack Scenario. In the second scenario we have the Byzantine processors acting as crashed processors. In fact, the Byzantine processors do not send or broadcast anything, expressly remaining completely idle. This is non-responsiveness scenario stresses the algorithm in the sense that all correct processors must participate in order to reach the $n - t$ thresholds.

Inverse Attack Scenario. In the next scenario, we have the Byzantine processors trying to attack the algorithm by sending the exact opposite values from those that are instructed by the protocol's specification. When a Byzantine processor needs to send a message, it modifies it before the transmission in an effort to confuse the correct processors and achieve a delay in the consensus process due to the fact that correct processors nodes might need to send more messages to agree on the binary value.

Half and Half Attack Scenario. This scenario has a similar approach as the previous one but this time the Byzantine processors do not send the same message to all processors. The Byzantine processors send the correct message to one half of the processors and a modified wrong message to the other half of the processors, in effort to hinder correct decision making.

Random Attack Scenario. The last Byzantine attack scenario is called "Random attack" because this time the Byzantine processors send messages with random values to other processors based on a probability p. Specifically, the Byzantine processors send different messages to every other processor and the value that contained in the message is randomly selected. For instance, when the probability $p = 0.25$ the Byzantine processors uniformly select one of the values $\{\}$, $\{0\}$, $\{1\}$, $\{0, 1\}$ to include in the message.

6.1.2 Transient faults injection

Corruption of state of processor. A transient fault can corrupt the state of processor $p_i \in P$ by, for example, setting $est[i]$ or $aux[i]$ with a different value set including the empty set.

Corruption of round counter. Another case of state corruption is when the round counter suddenly setting with a different value. Transient faults can cause the counter to either abruptly decrease or increase.

Corruption of a message. In-addition, transient faults can cause the message content to become corrupt. A corrupt message in transit may be received after several exchanges.

6.1.3 Combining failures

Depending on the experiment's aims some of experiments contain both types of failures, others only one type and others use no failures. The details are given in the dedicated descriptions of the experiments in the next section.

6.2 Experimental Environment

The evaluation was mainly performed in two different environments; a localhost environment run on a single machine and a more real-word distributed environment run on a cluster. This section introduces these environments and a description of how the evaluation was carried out along.

Localhost. A localhost environment was set up during the start of the project, to facilitate for fast prototyping and development. By not having to deploy the system to external servers prior to testing a new implementation, the overhead of implementation could be reduced and helped speed up the implementation. The basic CPU resource characteristics for the localhost machine are featured in Table 2. The local environment simulated multiple processors running on the same machine which all communicated with each other. This resulted in the system performing very well and reliable due to

Machine ID	**CPUs**	**Thread per core CPU**	**Cores per socket**	**Sockets**	**Model**	**CPU GHz**
0	8	1	4	2	Intel(R) Xeon(R) CPU	1.6

Table 2: Localhost machine basic CPU characteristics

Machine ID	**CPU(s)**	**Thread per core CPU**	**Core(s) per socket**	**Socket(s)**	**Model**	**CPU GHz**
0	8	1	4	2	Intel(R) Xeon(R) CPU	1.6
1	8	1	4	2	Intel(R) Xeon(R) CPU	1.6
5	1	1	1	1	AMD Opteron(tm) Processor 252	2.59
6	2	1	1	2	AMD Opteron(tm) Processor 246	1.99
8	1	1	1	1	AMD Opteron(tm) Processor 252	2.59

Table 3: Cluster machines' basic CPU characteristics

the minimal latency and overhead in message exchange, which was helpful to perform basic validation of the system quickly.

Cluster. In this case we used several machines in a local network that worked as our processors. Having them running in parallel and while connected through their public IP via a ZeroMQ socket API, the processors worked as a distributed system testbed. The machines we used were a cluster of five (5) hosts. Their basic CPU resource characteristics are featured in Table 3. On every physical machine we incremented the processes emulated by one in a round robin fashion. This allowed us to reach to 16 processes, where each machine ran 3 processes except one that ran 4.

6.3 Evaluation Criteria

A common evaluation criteria in the field is to measure the algorithm's latency. The average time it takes for an algorithm to decide on a single binary value is called *decision time*. This includes both communication delay and local processing time. We use this as our primary metric for the evaluation. In addition, it is important to measure the time it takes for the algorithm to recover from a transient fault. We, therefore, define the *converge time* to be the equivalent of the execution time it takes the algorithm to decide in a single binary value given that it starts in an illegal state, i.e., a corrupt state.

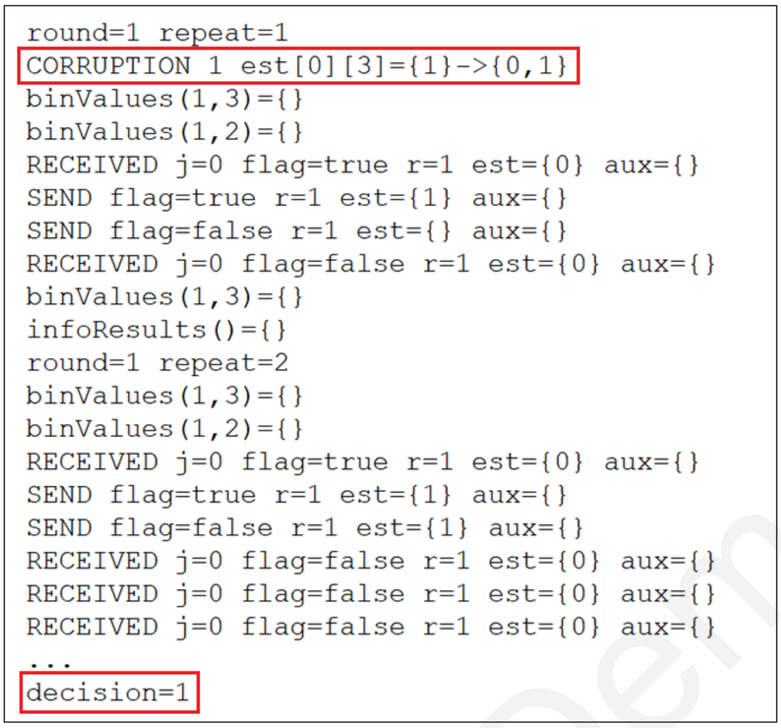
6.4 Experimental Validation and Evaluation

We performed experiments in two directions: (1) We performed an Experimental Validation of the algorithm's implementation. (2) We then proceeded to the Experimental Evaluation. In both case, we first started by running the experiments locally and then on the cluster.

6.4.1 Experimental Validation

For the experimental validation we ran several unit tests to check the correctness of the algorithm's implementation. A unit test is a small piece of code in our implementation that inject a fault so that we can validate the tolerance of the algorithm to faults. The unit tests performed are those described already in Section 6.1. Namely, we ran Byzantine failures (normal, idle, inverse, half&half, randomized), then corruptions by transient faults (corruption of state of processor, corruption of round counter, corruption of a message), and then combinations of the two. These experiments were run mostly locally since they concerned validation.

To perform the validation, a logger was implemented, which wrote the output and the errors in text files for the monitoring of the system's operation. We used the created log files to check that



```
round=1 repeat=1
CORRUPTION 1 est[0][3]={1}->{0,1}
binValues(1,3)={}
binValues(1,2)={}
RECEIVED j=0 flag=true r=1 est={0} aux={}
SEND flag=true r=1 est={1} aux={}
SEND flag=false r=1 est={} aux={}
RECEIVED j=0 flag=false r=1 est={0} aux={}
binValues(1,3)={}
infoResults()={}
round=1 repeat=2
binValues(1,3)={}
binValues(1,2)={}
RECEIVED j=0 flag=true r=1 est={0} aux={}
SEND flag=true r=1 est={1} aux={}
SEND flag=false r=1 est={1} aux={}
RECEIVED j=0 flag=false r=1 est={0} aux={}
RECEIVED j=0 flag=false r=1 est={0} aux={}
RECEIVED j=0 flag=false r=1 est={0} aux={}
...
decision=1
```

Figure 4: Example of a logger output for a processor that gets a transient error (corruption) and finally decides on a single binary value

the algorithm was exhibiting proper functionality per its specifications. This allowed us to identify minor bugs in the method, which we graciously informed the algorithm's authors about. For example, Figure 4 shows the logger output for a processor that gets a transient error (corruption). Suddenly a processor gets a transient fault that corrupts the state of processor. Due to the fact that the algorithm is self-stabilizing through the logs, we observe that the algorithm continues to execute based on its specifications. Finally, the processor decides on a single binary value (1) and the other processors decide on the same value.

6.4.2 Experimental Evaluation Results and Outcomes

The non-self-stabilizing implementation. Our experimental results were compared with the results of the non-self-stabilizing algorithm by Mostefaoui et al. [26] implemented as part of a degree project [27] under the same project umbrella. As already mentioned on Section 5.1, both implementations were developed in the same programming language (Go) and had the same communication layer (ZeroMQ). In addition, the same random bit function was used in an effort to ensure an as fair as possible experimental basis.

The decision time of each processor was measured from the moment that the algorithm starts its execution until the moment it decides on a single binary value. Then, to calculate the average decision time we deduced the average decision times of all non-faulty processors. To get more accurate results each experiment was repeated 10 times. We then had the maximal and minimal outlier of each experiment removed, and then calculated the average of the remaining values.

Self-stabilization overhead. Figure 5 and Figure 6 show the average decision time and the average number of messages received by a processor for the non-self-stabilizing, the self-stabilizing and an optimized self-stabilizing version of randomized BFT binary consensus algorithm without the presence of faults, on localhost and cluster respectively. The decision time and messages are measured by increasing the number of processors in the network.

More specifically, the decision time for the non-self-stabilizing algorithm increased slightly from 0.2 sec to 1.5 sec on the localhost and from 0.3 sec to 1.1 sec on the cluster, with the increase in the number of processors. In contrast, the decision time for the self-stabilizing algorithm increased significantly from 0.3 sec to 5.3 sec on the localhost and from 0.5 sec to 5.5 sec on the cluster, by the increase of the number of processors.

Trying to explain the overhead of self-stabilization, we decided to measure the average number of messages received by a processor that appear on Figure 5 and Figure 6 on a secondary axis. We notice that the decision time and the number of messages have the same trend, so we found that the overhead is due to the large number of messages sent during self-stabilization.

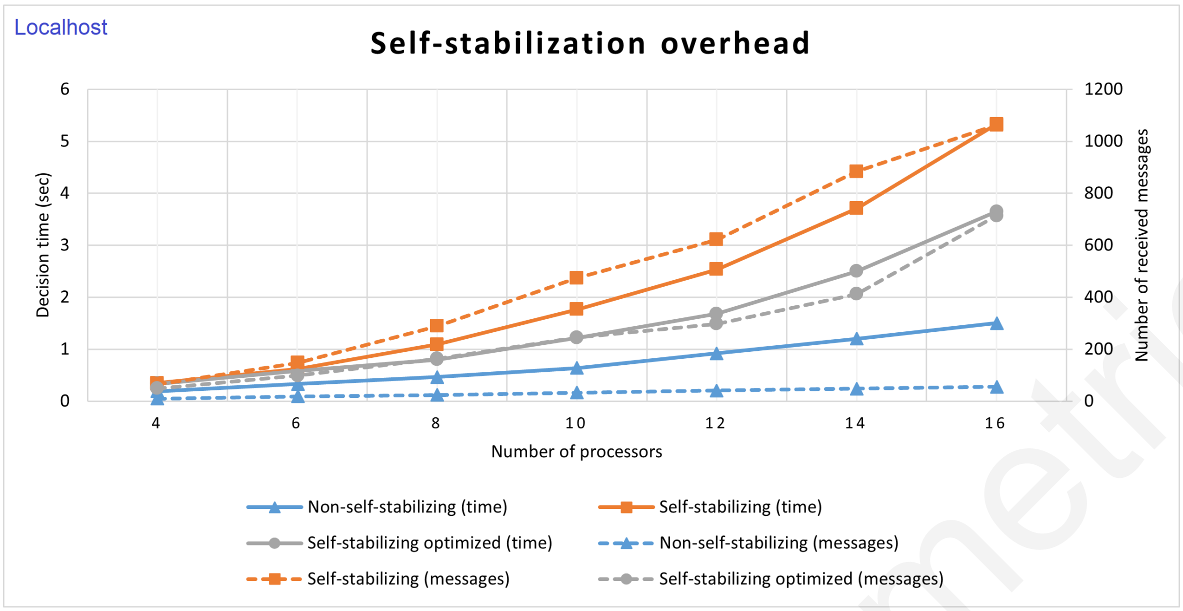


Figure 5: The average decision time and the average number of messages received by a processor for the non-self-stabilizing, the self-stabilize and an optimized self-stabilizing version of randomized BFT binary consensus algorithm without the presence of any type of faults. These experiments were performed on localhost.

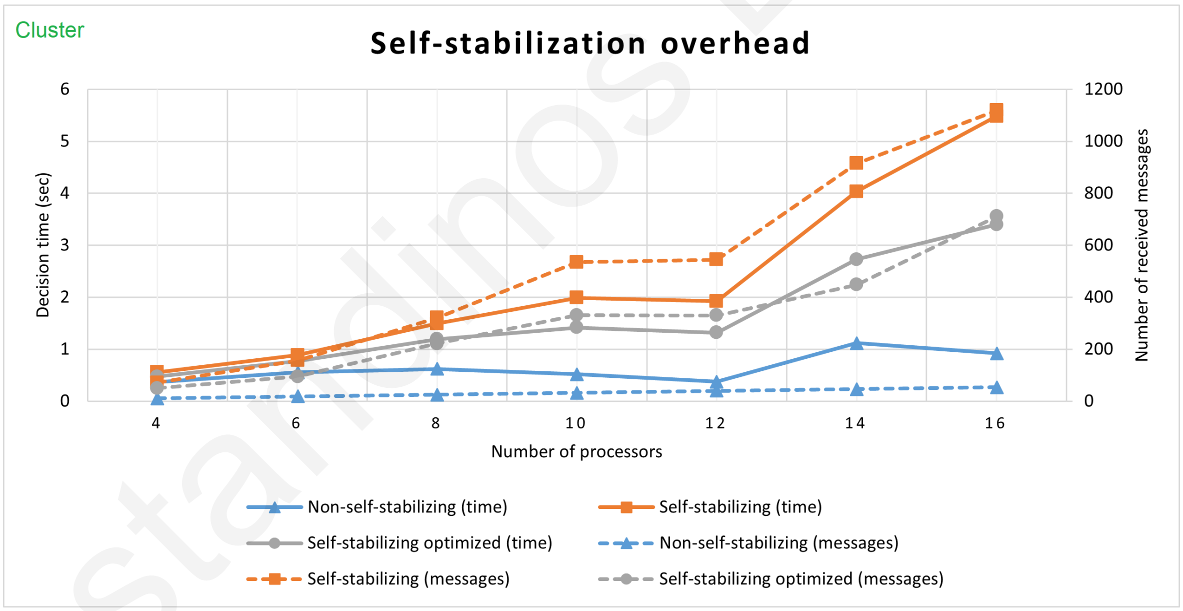


Figure 6: The average decision time and the average number of messages received by a processor for the non-self-stabilizing, the self-stabilize and an optimized self-stabilizing version of randomized BFT binary consensus algorithm without the presence of any type of faults. These experiments were performed on cluster.

An optimized self-stabilizing version. To check whether the decision time is affected by the number of messages received by each processor, we decided to create an optimized self-stabilizing version of the algorithm. The optimized self-stabilizing version is a variation of the initial version of self-stabilizing algorithm with the difference that instead of sending messages in each iteration, the messages the messages are sent periodically, and in particular in every 2nd iteration. Indeed, it turns out that this optimization improves the performance, but longer periods of sending messages did not contribute further reductions in time or messages needed to decide. We notice that the optimized version of the self-stabilizing algorithm presents lower overhead compared to the original version. More specifically, the decision time for the optimized self-stabilizing algorithm increased from 0.3 sec to 3.6 sec on the localhost and from 0.4 sec to 3 sec on the cluster, by the increase of the number of processors. Whether such an optimization is acceptable, depends on the application itself, since the longer the period between communications the longer it should be expected to correct a corrupt state. For example, if its operation is too critical, we may want to accept the overhead of sending messages in each iteration.

We also notice that in all versions of the algorithm that executed on cluster the decision time increases with the increase in the number of processors except in some cases where it seems that there is a slight decrease while we would expect it to increase. The reason that this happens is due to the hardware resource bottlenecks. Recall (Section 6.2) that we do not have enough machines in the cluster to run individual processes on each machine. Some processes are grouped on the same machine so these processes will exchange their messages faster so they will decide faster and thus reduce the average decision time. On the other hand, when all versions of the algorithm executed locally always the decision time increases with the increase in the number of processors, since all processors always run on the same machine.

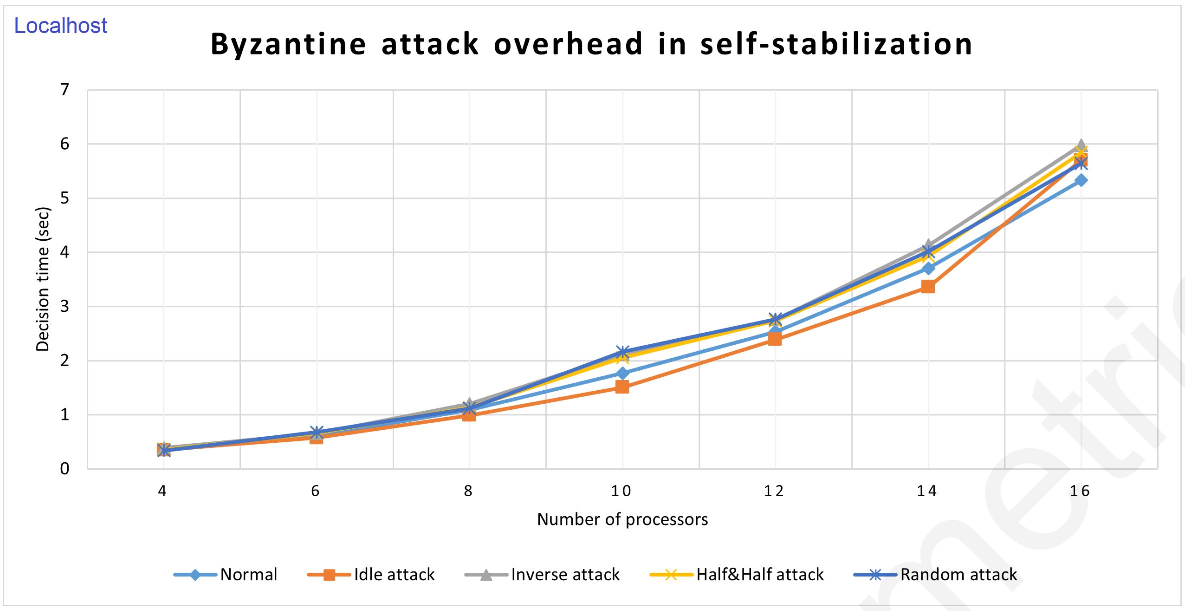


Figure 7: Decision time of the initial version of the self-stabilizing algorithm in the presence of Byzantine attacks. These experiments were performed on localhost.

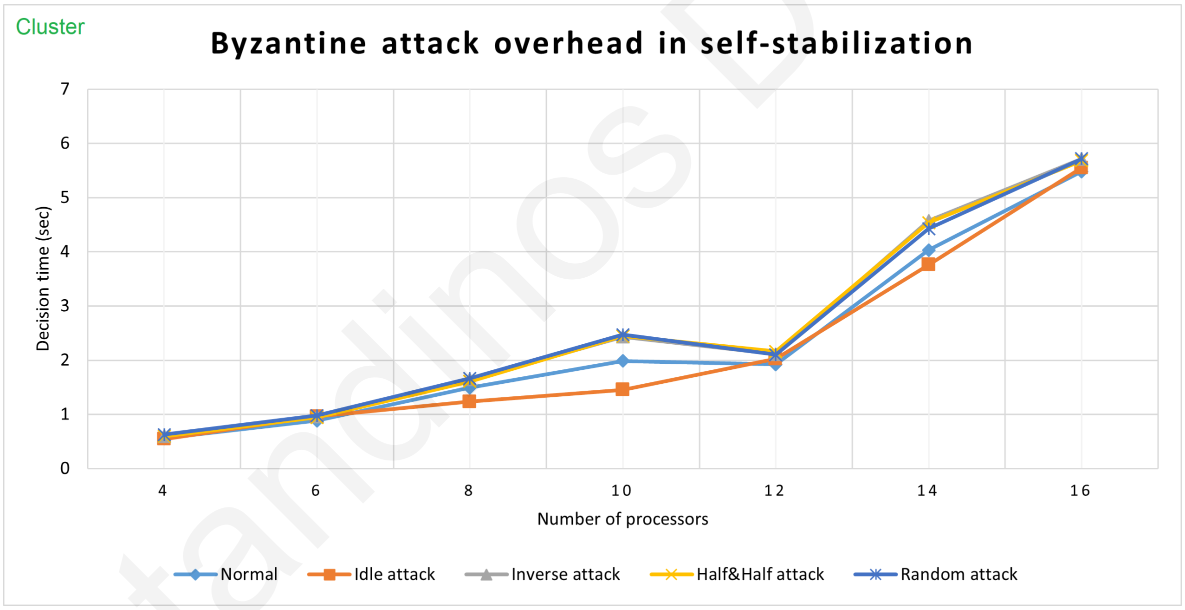


Figure 8: Decision time of the initial version of the self-stabilizing algorithm in the presence of Byzantine attacks. These experiments were performed on cluster.

Byzantine attack overhead. Figure 7 and Figure 8 illustrate the decision time of the initial version of the self-stabilizing algorithm in the presence of Byzantine faults as described in Section 6.1.1. More specifically, the decision time for the self-stabilizing algorithm in the present of Byzantine faults, increased from 0.38 sec to 5.9 sec on the localhost and from 0.6 sec to 5.7 sec on the cluster, for the most of Byzantine attacks, by the increase of the number of processors. We observe that the Byzantine faults do not cause significant overhead in the execution of the algorithm. The reason why decision time with idle attack is slightly less compared to normal execution (without Byzantine faults) is because the Byzantine processors do not send any messages, so the non-faulty processors have fewer messages to process. In other cases of Byzantine attacks the decision time is slightly increased since the Byzantine processors try to confuse the non-fault processors, ones so the non-fault processors have to do more iterations to decide a single binary value.

Figure 9 and Figure 10 illustrate the decision time of the non-self-stabilizing algorithm in the presence of Byzantine faults. The first thing to note is that the Byzantine faults do not cause significant overhead in the execution of the non-self-stabilizing algorithm. More specifically, the decision time for the self-stabilizing algorithm in the present of Byzantine faults, increased from 0.2 sec to 1.8 sec on the localhost and from 0.4 sec to 1.1 sec on the cluster, for the most of Byzantine attacks, by the

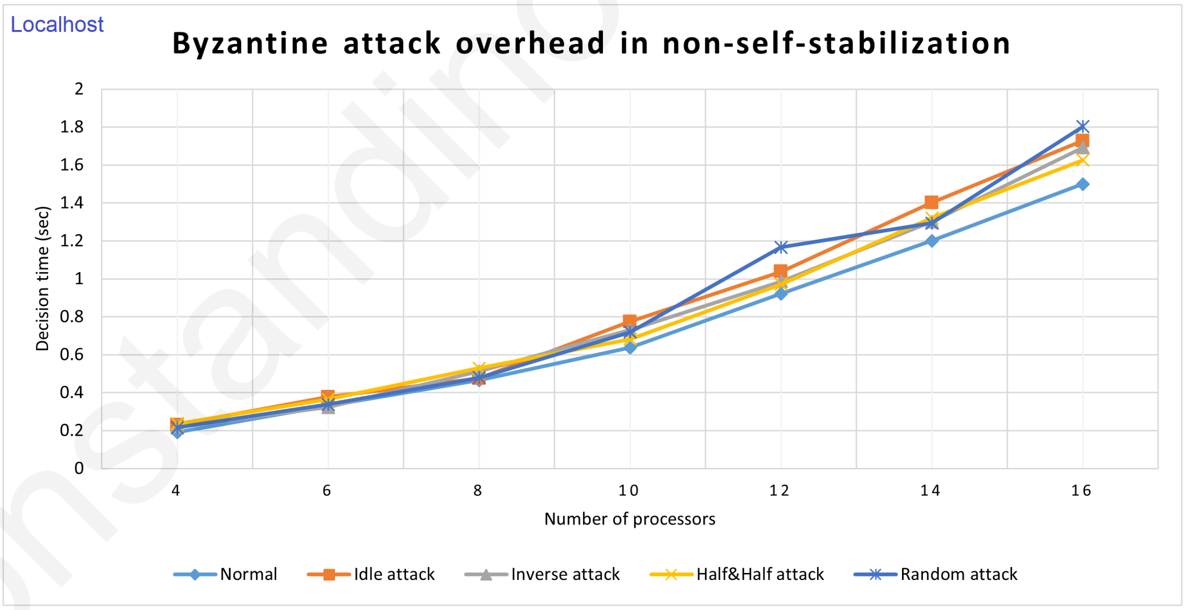


Figure 9: Decision time of the non-self-stabilizing version of the algorithm in the presence of Byzantine attacks. These experiments were performed on localhost.

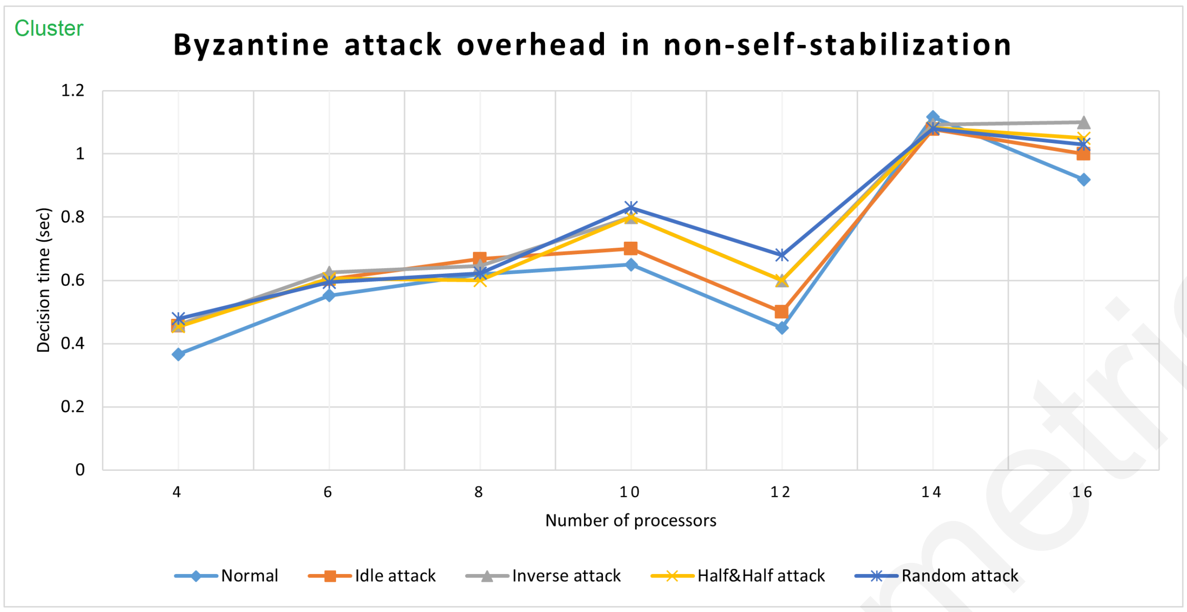


Figure 10: Decision time of the non-self-stabilizing version of the algorithm in the presence of Byzantine attacks. These experiments were performed on cluster.

increase of the number of processors. In other words, non-self-stabilizing algorithm behaves similarly to the self-stabilizing algorithm in the presence of Byzantine faults. We also notice that in all types of Byzantines attacks the decision time on cluster, sometimes it is slight decrease while we would expect it to increase. The reason that this happens is due to the fact that we do not have enough machines in the cluster to run individual processes on each machine.

Convergence overhead. Finally, Figure 11 and Figure 12 show the converge time of the initial version of the self-stabilizing algorithm. As mentioned in Section 6.3, it is important to measure the time it takes for the algorithm to recover from a transient fault. We, therefore, define the converge time to be equivalent to the execution time it takes the algorithm to decide, given that it starts in an illegal state (state with transient faults). We observe that convergence time is approximately the same as decision time without transient faults which means that the convergence overhead is not significant. More specifically, the convergence time for the self-stabilizing algorithm increased slightly from 0.4

sec to 5.4 sec on the localhost and from 0.5 sec to 5.1 sec on the cluster, with the increase in the number of processors.

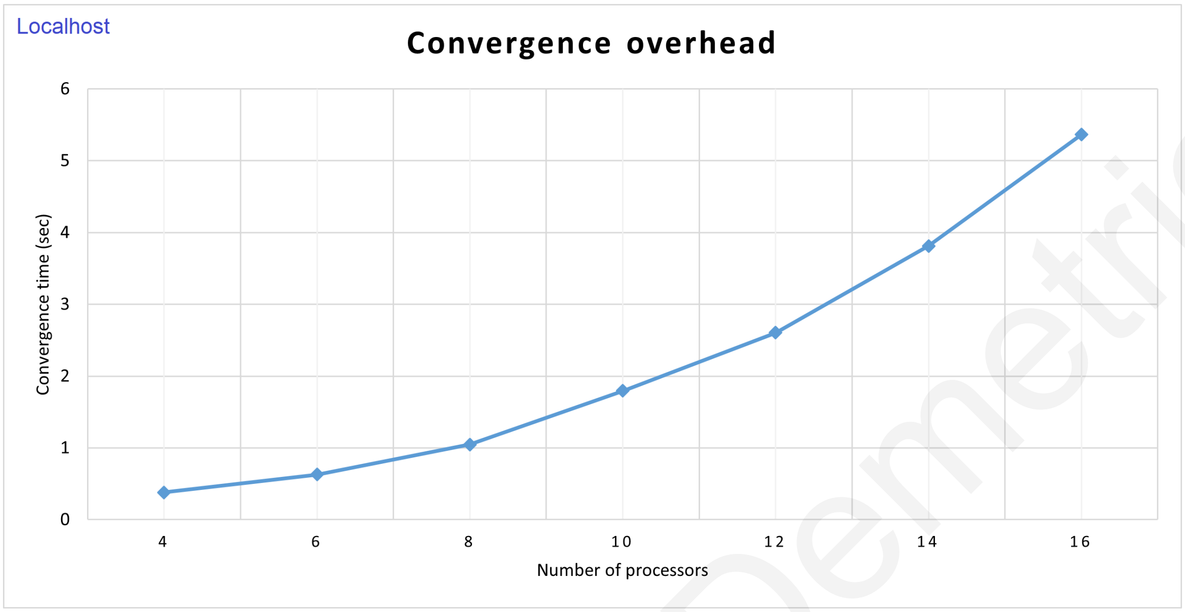


Figure 11: Converge time of the initial version of the self-stabilizing algorithm. These experiments were performed on localhost.

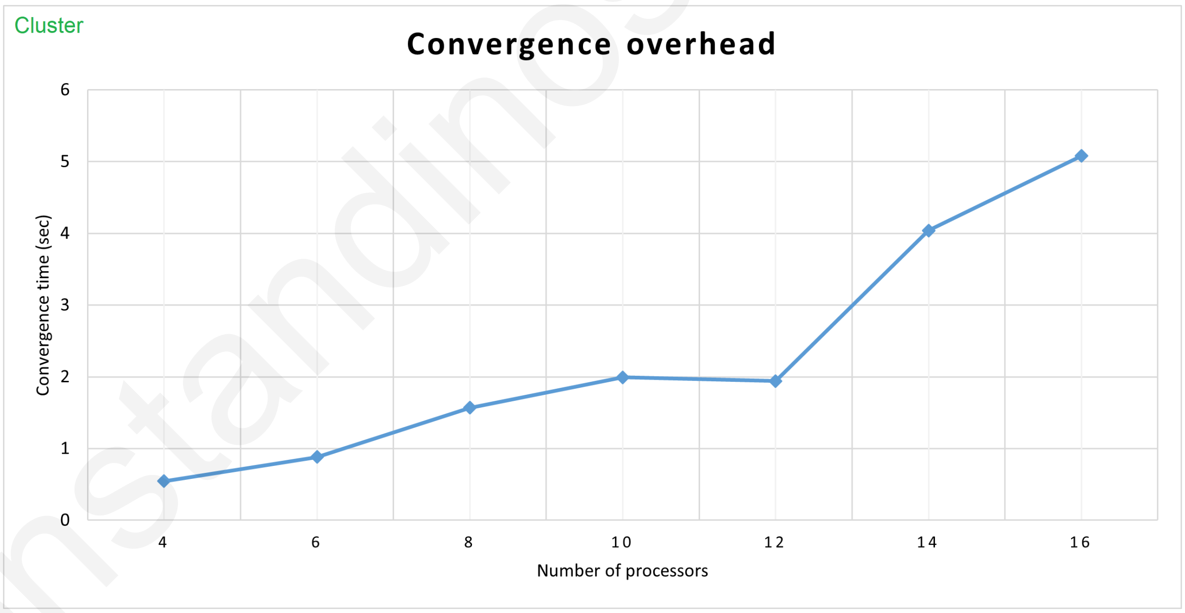


Figure 12: Converge time of the initial version of the self-stabilizing algorithm. These experiments were performed on cluster.

Chapter 7

Conclusions

In last chapter, we present a brief summary of our work and the challenges we faced and suggest future work that could be done.

7.1 Summary

Binary consensus is a fundamental problem in distributed systems that is especially hard to solve for some system models. Our case-study is, to the best of our knowledge, the first work to practically implement and evaluate a self-stabilizing randomized Byzantine fault-tolerant binary consensus algorithm. For the implementation we use the Go programming language together with the ZeroMQ message-passing library.

In this thesis, we explore binary consensus for a specifically complicated failure model; processors may exhibit Byzantine (malicious) behavior by not following the algorithm's specifications. An arbitrary transient fault represents any possible temporary violation of the assumptions that can happen to a system, except that the algorithm code stays intact.

The implementation served a validation of the algorithm's correctness using several unit tests. By checking the logs files generated by the algorithm we have found that the implemented algorithm can tolerant Byzantine faults and it can recover from transient faults effectively.

In addition, we proceeded to the experimental evaluation with respect to the performance. We first started by running the experiments locally and then on the cluster. Experimentally we find that for the self-stabilization there is some overhead. Therefore, we decided to develop an optimized variation of the initial version of self-stabilizing algorithm with the difference that instead of sending messages in each iteration, the messages the messages are sent periodically. A transient fault under normal circumstances happens very rarely, so we do not want to burden the system with unnecessary messages for the self-stabilization. However, it should be emphasized that the decision is based mainly on the application. If its operation is too critical, we may want to accept the overhead of sending messages in each iteration. Our last finding is related to the fact that Byzantine and transient corruptions do not cause significant overhead in the performance of the algorithm.

7.2 Challenges

The main challenge in the implementation was to establish asynchronous communication. We had to avoid the case where a processor might block waiting for another processor's response. Even though REQ/REP sockets are synchronous, we used them due to the fact that we wanted to have a common communication layer with the non-self-stabilizing algorithm. We solved this challenge by using Go goroutines, channels and the `select` statement alongside the `timeticker` functionality that Go offers.

Furthermore, we have never worked with the Go programming language and the ZeroMQ library (distributed programming in general). The procedure gave us a better understanding of these two state-of-the-art concepts. The whole process of studying and coding the implementation took around one month of dedicated effort. Another difficulty was debugging the system, since in a distributed system is much more challenging than debugging a single processor program, let alone debugging a distributed system with concurrency.

On the experimental side, we highlight that the number of resources were limited due to the number of machines in the cluster and the low specifications of the machines. We had only five (5) machines at our disposal, which limited our ability to examine how the algorithm reacts on a larger scale network.

7.3 Future Work

Future work could take on multiple dimensions. More optimization could be used to make the algorithm even more competitive. We can look for more ways (besides periodic messaging) by which we can reduce the execution time of the algorithm.

On the experimental side, we note that there were limitations in the number of resources, as the number of machines in the cluster we used and the low specs of the machines. We will be able to add to the existing cluster more machines with higher specifications. This will avoid the possibility of more than one processor running on the same machine, which affected the average decision times.

Finally, it is interesting to evaluate how the algorithm reacts on a larger scale network. Therefore, experiments on a larger scale platform such as AWS or Planet-lab will reveal trends that we were not able to see within our smaller scale cluster. In real large-scale networks, the processors will be geographically remote so we may be experiencing events such as network delays or packet loss that are unlikely to occur on the local network where the cluster is located. Hence, possibly our synchrony emulation with REQ/REP sockets would no longer be effective, so the implementation should be modified to use the DEALER/ROUTER paradigm.

Bibliography

[1] [n.d.]. Gob. https://pkg.go.dev/encoding/gob Accessed: 2022-05-29.

[2] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. 1999. Revising the Weakest Failure Detector for Uniform Reliable Broadcast. In *Distributed Computing, 13th International Symposium, Bratislava, Slovak Republic, September 27-29, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1693)*, Prasad Jayanti (Ed.). Springer, 19–33. https://doi.org/10.1007/3-540-48169-9_2

[3] James Aspnes. 2003. Randomized protocols for asynchronous consensus. *Distributed Comput.* 16, 2-3 (2003), 165–175. https://doi.org/10.1007/s00446-002-0081-5

[4] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. 2008. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, Rida A. Bazzi and Boaz Patt-Shamir (Eds.). ACM, 385–394. https://doi.org/10.1145/1400751.1400802

[5] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.* 18, 3 (2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[6] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria (LIPIcs, Vol. 91)*, Andréa W. Richa (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:16. https://doi.org/10.4230/LIPIcs.DISC.2017.1

[7] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[8] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (1996), 225–267. https://doi.org/10.1145/226643.226647

[9] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *Comput. J.* 49, 1 (2006), 82–96. https://doi.org/10.1093/comjnl/bxh145

[10] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. 2011. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. J. Crit. Comput. Based Syst.* 2, 2 (2011), 141–161. https://doi.org/10.1504/IJCCBS.2011.041257

[11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design*. Pearson.

[12] Ariel Daliot and Danny Dolev. 2006. Self-stabilizing byzantine agreement. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, Eric Ruppert and Dahlia Malkhi (Eds.). ACM, 143–152. https://doi.org/10.1145/1146381.1146405

[13] Constandinos Demetriou. [n.d.]. constandinos/self-stabilizing-binary-consensus: Self-stabilizing randomized Byzantine-tolerant binary consensus. https://github.com/constandinos/self-stabilizing-binary-consensus Accessed: 2022-05-28.

[14] Edsger W. Dijkstra. 1974. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* 17, 11 (1974), 643–644. https://doi.org/10.1145/361179.361202

[15] Shlomi Dolev. 2000. *Self-Stabilization*. MIT Press.

[16] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. 2018. Self-stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors. In *Cyber Security Cryptography and Machine Learning - Second International Symposium, CSCML 2018, Beer Sheva, Israel, June 21-22, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10879)*, Itai Dinur, Shlomi Dolev, and Sachin Lodha (Eds.). Springer, 84–100. https://doi.org/10.1007/978-3-319-94147-9_7

[17] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. 2021. Self-stabilizing Byzantine- and Intrusion-tolerant Consensus. *CoRR* abs/2110.08592 (2021). arXiv:2110.08592 https://arxiv.org/abs/2110.08592

[18] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[19] Chryssis Georgiou, Ioannis Marcoullis, Michel Raynal, and Elad Michael Schiller. 2021. Loosely-self-stabilizing Byzantine-Tolerant Binary Consensus for Signature-Free Message-Passing Systems. In *Networked Systems - 9th International Conference, NETYS 2021, Virtual Event, May 19-21, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12754)*, Karima Echihabi and Roland Meyer (Eds.). Springer, 36–53. https://doi.org/10.1007/978-3-030-91014-3_3

[20] go [n.d.]. The go programming language. https://golang.org/. Accessed: 2022-05-12.

[21] Pieter Hintjens. 2013. *ZeroMQ: Messaging For Many Applications* (first ed.). O'Reilly Media.

[22] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

[23] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. 2021. Self-Stabilizing Indulgent Zero-degrading Binary Consensus. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*. ACM, 106–115. https://doi.org/10.1145/3427796.3427836

[24] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. 2021. Self-stabilizing Multivalued Consensus in Asynchronous Crash-prone Systems. In *17th European Dependable Computing Conference, EDCC 2021, Munich, Germany, September 13-16, 2021*. IEEE, 111–118. https://doi.org/10.1109/EDCC53658.2021.00023

[25] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[26] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. 2014. Signature-free asynchronous Byzantine consensus with t 2<n/3 and o(n^2) messages. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, Magnús M. Halldórsson and Shlomi Dolev (Eds.). ACM, 2–9. https://doi.org/10.1145/2611462.2611468

[27] Vassilis Petrou. 2021. Implementation and Empirical Evaluation of a Randomized Byzantine Fault-Tolerant Distributed Algorithm. (May 2021). Bachelor Thesis. University of Cyprus, Department of Computer Science.

[28] randomfunc [n.d.]. rand. https://pkg.go.dev/math/rand. Accessed: 2022-05-12.

[29] Michel Raynal. 2018. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer. https://doi.org/10.1007/978-3-319-94141-7

[30] Marco Schneider. 1993. Self-Stabilization. *ACM Comput. Surv.* 25, 1 (1993), 45–67. https://doi.org/10.1145/151254.151256

[31] Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. 2012. Loosely-stabilizing leader election in a population protocol model. *Theor. Comput. Sci.* 444 (2012), 100–112. https://doi.org/10.1016/j.tcs.2012.01.007

[32] Junko Yoshida. 2022. Toyota case: Single Bit Flip that killed. https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/ Accessed: 2022-05-20.

[33] zeromqSite [n.d.]. ZeroMQ. https://zeromq.org/. Accessed: 2022-05-12.

[34] zeromqSocket [n.d.]. ZeroMQ Socket API. https://zeromq.org/socket-api/. Accessed: 2022-05-12.

[35] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. 2017. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In *2017 IEEE International Congress on Big Data, BigData Congress 2017, Honolulu, HI, USA, June 25-30, 2017*, George Karypis and Jia Zhang (Eds.). IEEE Computer Society, 557–564. https://doi.org/10.1109/BigDataCongress.2017.85