Bachelor Thesis

IMPLEMENTATION, VALIDATION AND EXPERIMENTAL EVALUATION OF A SELF-STABILIZING BYZANTINE FAULT-TOLERANT ATOMIC BROADCAST ALGORITHM

Giorgos Evangelou

UNIVERSITY OF CYPRUS

Department of Computer Science

June 2022

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Implementation, Validation and Experimental Evaluation of a Self-stabilizing Byzantine Fault-Tolerant Atomic Broadcast Algorithm

Giorgos Evangelou

Advisor

Chryssis Georgiou

This diploma project has been submitted for partial fulfillment of the requirements of Computer Science Degree acquisition from the University of Cyprus

June 2022

Acknowledgement

Firstly, I would like to share my sincere gratitude to my advisor Professor Chryssis Georgiou, who gave me the opportunity to work on this fascinating project. He pushed me to achieve the best possible results throughout this year and helped me learn many new things. Moreover, I want to share my appreciation for the assistance of Dr. Ioannis Marcoullis for the clarification of gray areas of the project. Our collaboration was superb, and both provided to me excellent guidance during this year.

Furthermore, this thesis was partially 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”.

For the accomplishment of this project, I would also like to thank my family and friends who have been by my side all these years. Their influence was critical for the achievement of this thesis. Additionally, I want to thank them for their support in every step of my life, which motivates me to continue pursuing my dreams.

Abstract

In distributed systems, Byzantine Fault Tolerance (BFT) is essential for normal operation as BFT algorithms offer protection in several situations, such as when having crashed or malicious processes in the system. BFT algorithms, however, are not immune to transient faults, which can alter the state of the system unexpectedly and lead it to an unsafe one. Due to this, over the last few years, the development of *self-stabilizing* algorithms, which can automatically recover from transient faults, has gained a significant role in the research community.

Moreover, total order reliable broadcast, namely *atomic broadcast*, is a fundamental building block in several real-life applications. Any distributed system that needs to store the same states of its objects across its processes, requires the atomic delivery of the states of these items. Therefore, the increased popularity of distributed environments with the necessity of state machine replication leads to a rise in the usage of atomic broadcast protocols.

In this thesis, a stack of new self-stabilizing Byzantine-resistant algorithms is implemented and validated. Namely, a known stack of protocols that has been transformed to self-stabilizing was evaluated for its behavior. This stack consists of vector consensus and atomic broadcast protocols with each of them operating in an asynchronous decentralized environment. Both are optimal in resilience as they can tolerate up to f faulty nodes out of n, where $f < n/3$.

The deployment of the above-mentioned algorithms is completed in the Go programming language by using the ZeroMQ library for message exchanges. The implementation was used to evaluate the algorithms' correctness and convergence in several scenarios. These validation unit tests consist of different environment configurations based on the existence of failures. Specifically, various tests took place with the presence of Byzantine processes, transient faults, or the combination of these failures. The protocols were assessed in fault-free scenarios, too, where every node in the system corresponds to a correct process.

Furthermore, one of the goals of this thesis is to present an experimental evaluation of the proposed algorithms. That is, to measure their performance beyond the theoretical scope. These evaluations include scenarios with different number of system processes and unit tests to assess the ability of the protocols to tolerate Byzantine nodes and transient faults. In addition, convergence time experiments took place to find the required time of the algorithms to converge to a safe state from an arbitrary one. The experiments indicate the additional overhead of self-stabilization.

Contents

Notable Abbreviations

- **ABC**: Atomic Broadcast
- **ACK**: Acknowledge (message type in computer networking)
- **BC**: Binary Consensus
- **BFT**: Byzantine Fault Tolerance
- **BVB**: Binary-value Broadcast
- **CSP**: Communicating Sequential Processes
- **f**: number of Byzantine nodes in a system
- **FLP Impossibility Result**: Fischer-Lynch-Patterson Impossibility Result
- **MVC**: Multi-valued Consensus
- **n**: total number of processes in a system
- **PBFT**: Practical Byzantine Fault Tolerance
- **PK**: Public Key
- **RBC**: Reliable Broadcast
- **SSABC**: Self-stabilizing Atomic Broadcast
- **SK**: Secret Key
- **SSVC**: Self-stabilizing Vector Consensus
- **SMR**: State Machine Replication
- **VC**: Vector Consensus

Chapter 1

Introduction

1.1 Motivation

The agreement of several distributed processes on some values proposed by them is essential for various real-world applications, such as state machine replication, atomic broadcast and more broadly blockchains and cloud computing systems. The addition of crash-prone or malicious processes makes the achievement of system reliability more difficult. This is defined as the consensus problem [20, 43], where the correct processes, out of the n total, must agree on a value based on their proposals, despite the existence of f Byzantine ones that exhibit arbitrary behavior. Each non-faulty process proposes a value and every one of them decides the same value. As a result of its significance, the research community devoted much effort to study the consensus problem and its applications [16, 28, 40].

Regarding consensus, Byzantine Fault Tolerance (BFT) can be achieved in synchronous environments as long as the number of Byzantine processes, f, is less than a third of the overall system processes, n, where $f < n/3$ [34]. In asynchronous systems, however, the FLP impossibility [23] implies that consensus is impossible even with one crashed process. Researchers use different approaches to circumvent the FLP, such as synchronous assumptions or randomized methods either with the usage of cryptographic methods [21, 30] or not [15].

In addition, the state of a system could be altered as a virtue of the existence of transient failures. Transient faults can corrupt the local state of an algorithm, like messages in communication channels or its memory. Therefore, this is a critical flaw to be considered in algorithmic thinking, as the results of transient faults could be disastrous. Distributed algorithms can be designed to be *self-stabilizing*, which means that if they deviate from a correct state, due to a transient fault, they always converge to a correct one in finite steps. Despite BFT consensus being a well-studied problem, there are not many self-stabilizing solutions.

Consequently, the creation and study of self-stabilizing BFT algorithms for the consensus problem is essential for a wide range of systems, because they offer very strong safety guarantees against various faults. The future of technology could be brighter, as it would excel by the development of real-world applications based on these principles.

1.2 Contributions

The goal of this thesis is to implement, and experimentally validate and evaluate self-stabilizing Byzantine fault tolerant randomized algorithms for the Vector Consensus and Atomic Broadcast problems, which are based on the corresponding non-self-stabilizing versions of Correia et al. [15]. In detail, the algorithms developed were validated for their correctness and their applicability to real world scenarios as they were executed in real world-like simulations. Comparisons between self-stabilizing algorithms and their non-self-stabilizing counterparts took place to assess the difference in execution time and performance in general.

1.3 Methodology

At first, a background study took place to get more familiar with terms, principles, and problems of distributed computing. The study was mainly around fault tolerance by learning about the consensus and atomic broadcast problems [15, 20, 43], state machine replication [8], Byzantine fault tolerance [8, 12, 15, 20], and self-stabilization [17, 18, 19, 37]. Next, an in-depth study of the BFT consensus algorithms stack of Correia et al. [15] followed, to understand the basis of the creation and development of the corresponding self-stabilizing versions of vector consensus and atomic broadcast. Afterwards, a study of the implementation of the system we use took place, which was developed in an older thesis [35], and lead to the understanding of the simulation environment for several experiments that would take place. In addition, a learning period was required to practice the syntax and programming of the Go programming language's [39] scripts, as well as the usage of the ZeroMQ [42] messaging library to exchange information between processes.

The Agile Software Development process was applied throughout the implementation period. The focus of this process was the functionality of the system, while continual improvement and implementation changes took place to fully cover the possible execution scenarios. During this period, many meetings with the team of the Efficient Self-stabilizing Byzantine Fault Tolerance (ESS – BFT) project [22] were held to decide on crucial details of the algorithms. In addition, various debugging tests were used to find possible bugs and logical algorithmic errors. Succeeding the implementation phase, a lot of unit tests and experiments were applied to check the algorithms' validity. Moreover, their performance was evaluated in real-world situations, more specifically in a five-machine cluster. Finally, comparisons were made based on the measurements of the newly developed self-stabilizing algorithms and their non-self-stabilizing counterparts.

1.4 Thesis Structure

This thesis continues with the following chapters:

- **Chapter Two**: The background required for the formation of this thesis' algorithms regarding fault tolerance, consensus, self-stabilization, the Go language and the ZeroMQ library.
- **Chapter Three**: An overview of the original and the new self-stabilizing BFT atomic broadcast stacks.
- **Chapter Four**: A detailed presentation of the algorithms' implementation and specific decisions that had to be made.
- **Chapter Five**: Description of the experimental environment used with reference to the testing scenarios that took place.
- **Chapter Six**: Conclusions of this thesis and suggestions on future work to be done.

Chapter 2

Background

2.1 State Machine Replication

Distributed systems are a crucial part of modern Internet infrastructure as they contribute to many applications. These systems ought to be fault-tolerant as system failures of any form in software, hardware (e.g., power failures, cyber-attacks) have severe results and stop the system from operating smoothly. One technique to tackle the fault-tolerance problem is *redundancy*. Redundancy based systems have distributed objects copied in multiple processes to provide scaling and to hide processor failures in the overall system. However, this design introduces the problem of *consistency* as every replica of each object must exist in the same state in every process. *State machine replication* (SMR) [10] is one of the suggested methods that preserve the consistency of the replicated items. The state of the machine is determined by a set of inputs, known as *transactions*. More precisely, transactions are atomic operations, therefore they cannot be in an intermediate state [25]. This means that they are either complete or do not happen at all. SMR shares identical state transitions of an object's copy to ensure that object is similarly maintained in every process of the system.

2.2 Asynchrony

Safety and liveness are two aspects of a system that must be preserved for smooth operation. *Safety* implies that nothing bad happens, while *liveness* states that something good will occur eventually. Violation of safety can get a system to an undesirable state. For instance, one could get multiple different valid transaction logs and a non-responding system, by violating safety. In synchronous environments with knowledge of the maximum processor clock speed or message delay, it is trivial to decide for the transactions to be completed. This could happen by voting in each round, which is determined by the maximum time bounds of the system.

In asynchronous systems, information about processor speeds or message delays does not exist. Thus, it is a lot harder to satisfy safety and liveness in asynchronous environments. In addition, the FLP impossibility result illustrates that even with failure of one process, distributed consensus among processes in an asynchronous environment is not possible [23]. The FLP impossibility can be overcome by either applying more strict synchrony constraints, with coordinating leaders and timeouts, or by introducing non-deterministic components that have high probability of achieving consensus.

2.3 Broadcast and Consensus

Reliable Broadcast (RBC) is an important broadcast functionality which enables information delivery to achieve SMR. Basically, RBC ensures that a message will be delivered on every correct process and satisfies the following properties for a message *m* [8, 13]:

- **Validity**: every correct process that broadcasts *m*, will eventually deliver *m*.
- **Agreement**: if a correct process delivers *m*, all correct processes eventually deliver *m*.
- **Integrity**: the delivery of *m* happens only once, and only if broadcast by its sender.

Atomic Broadcast (ABC) is a stricter primitive, which satisfies total order in addition to RBC's properties [13]. **Total order** indicates that if two correct processes *p* and *q* deliver *m* and *m'*, then *p* delivers *m* before *m'* if and only if *q* delivers *m* before *m'*. In essence, ABC is a reliable broadcast which guarantees that messages are delivered to every process in identical order.

The following properties are satisfied by the consensus primitive [8, 13]:

- **Termination**: each correct process eventually decides.
- **Integrity**: each correct process decides at most once.
- **Agreement**: if one correct process decides v and another decides u, then $v = u$.
- **Validity**: if a correct process decides u, at least one correct process proposed u.

Principally, consensus is the agreement of a system's processes, which will eventually happen, on a specific value.

In general, a hierarchy of protocols from binary consensus to atomic broadcast exists and is presented in Figure 2.1. Upper layer primitives can be built by using functionality from the previous layer. Binary and multi-valued consensus are procedures which provide agreement on a value from a set of possible values of size 2 and greater than 2, respectively. *Vector consensus* ensures that a predetermined vector will be filled with values in every position with processors agreeing on the values placed. Hence, multi-valued consensus can be applied to fill vector positions. In atomic broadcast, correct processes agree on specific values and their position in the overall placement sequence (transaction log).

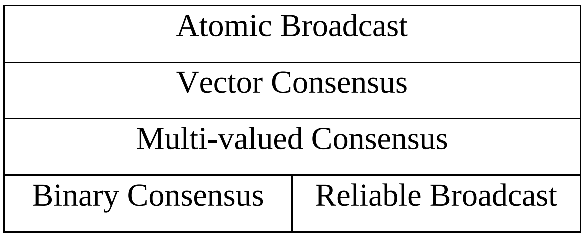


Figure 2.1: Binary consensus to atomic broadcast hierarchy

2.4 Crash Fault Tolerance

The general objective of a fault-tolerant system is to continue operating as intended despite having failures in some processes [38]. One category of malfunctions in distributed computing is crash failures. In a crash fault a process halts, hence it cannot lie or act maliciously to other processes. Crash fault tolerance is enforced by achieving consensus in a majority vote way. This is possible if the processes of a system are at least $2f + 1$, where f

indicates the maximum number of crashed machines. As previously mentioned, in asynchronous systems deterministic agreement is impossible if a processor fails [23].

2.5 Byzantine Fault Tolerance

Byzantine faults make a process to behave undesirably by not executing the defined algorithm and it is the most critical kind of failure in distributed systems. Both non-intentional faults and malicious ones are included in this category. Synchronous environments can achieve consensus as long as the total number of processes in the system is at least $3f + 1$, where f is the maximum number of Byzantine processors allowed [34].
Byzantine Fault Tolerance (BFT) can be achieved by either solving Byzantine atomic broadcast or Byzantine consensus. Byzantine atomic broadcast ensures that all correct processes in a system will receive the same set of messages in the same sequence or all of them will abort the delivery without consequences. Byzantine consensus defines the agreement of a value by the correct processes of the system, despite the existence of malicious ones. Both BFT leading directions are equivalent [5, 31], thus the FLP impossibility holds for Byzantine atomic broadcast, too, as it was introduced for the Byzantine consensus problem. Liskov and Castro suggested Practical Byzantine Fault Tolerance (PBFT) [12], which is a consensus BFT algorithm in asynchronous environments. This approach achieves consensus by using the majority rule with nodes being sequentially ordered and one of them being the primary – leader node, whereas the remaining ones are marked as secondary. Several assumptions are required for PBFT to operate appropriately. Namely, the following are necessary for PBFT:

i. the number of Byzantine processors must be at lower than a third of the total processes of the system
ii. cryptographic mechanisms availability
iii. unbounded local memory
iv. consistent initial state.

2.6 Randomized Byzantine Fault Tolerance

Randomized algorithms have great impact on a variety of topics in computer science due to their simplicity and their loose time expectations [36]. Moreover, they offer the only known polynomial solution to specific problems. Randomized agreement to tackle the FLP impossibility without the need of timing speculations was introduced by Rabin [36] and Ben-

Or [5]. These works inspired many papers with similar approach [1, 2, 10], although they lack efficiency as they have high computational complexity and communication as a virtue of the applied cryptography and many communication steps [32]. Another suggestion was proposed by Cachin et al. [9], which achieves BFT atomic broadcast with the use of coin-tossing, digital signatures, and public-key cryptography. Overall, protocols that use probabilistic decisions are called optimal in resilience if they are fault-tolerant when the total number of processes in the system is $3f+1$, with f being the number of faulty machines.

2.7 Self-stabilization

Fault-tolerant systems are crucial to overcome processes' failures of any kind, although they operate based on special assumptions. For instance, the number of failed processes will not necessarily be strictly within the predetermined bounds of the system during operation, or fortuitous bit flips may occur, which make the system unresponsive. These events fall into the category of *transient failures*, which are temporary, and can affect the system state but not its behavior. Despite their rare occurrence or their limited lifespan, transient faults can lead a system to an arbitrary state where recovery is impossible without human intervention.
Self-stabilizing models are able to cope with transient failures and can recover the system to a legitimate state as long as they do not occur continuously. Formally, a system S is self-stabilizing with respect to a predicate P, which signifies the proper execution of the system, if it satisfies the following [3, 37]:

- **Convergence**: starting from any possible system state, S will converge to a safe state where P is satisfied, in finite system state transitions.
- **Closure**: if the system is in a safe state satisfying P, then P cannot be falsified by the proper execution of S.

In general, states that satisfy predicate P are known as *legitimate* or *safe*, whereas the arbitrary ones which do not assure P are called illegitimate or unsafe.

2.8 Self-stabilizing BFT

The self-stabilizing BFT problem was approached by different works with dissimilar assumptions. Dolev et al. [18] check for malicious behavior of the PBFT leader [12] by using failure detection. The global state of the system is also monitored and when a transient fault is detected, the system transitions to a default state. It is important to mention that recognizing the difference between arbitrary state and malicious messages' corruption is extremely difficult for a process in an inconsistent state. Binun et al.s' [6] approach is based on semi-synchronous environments and assures Byzantine consensus in every clock pulse by using a self-stabilizing BFT clock synchronization algorithm [17].

A common factor of several solutions for BFT, i.e., Byzantine consensus and atomic broadcast, is the use of cryptographic procedures, which are not self-stabilizing. The existence of private keys that could be altered by transient faults make the system ineffective in transient-prone environments. This increases the difficulty of self-stabilization algorithms' design. Usually, cryptography for authenticating the sender of a message is replaced by having trusted bidirectional communication channels between processes, which ensure that malicious processes cannot send messages that seem to have other processes as senders.

Overall, the amount of research works that try to solve the self-stabilizing BFT problem are only a few.

2.9 Randomized Self-stabilization

Randomization and self-stabilization are important design factors that have seen substantial focus on various works. However, the difference between self-stabilizing randomized algorithms and randomized self-stabilizing ones should be mentioned. In the first kind, protocols included are deterministic in their convergence, while often probabilistic in termination. The second ones converge with high chance, so they stabilize probabilistically.

Randomized self-stabilizing algorithms are defined by either of the following:

- **Probability-based**: protocol converges to a safe state with probability 1 [26].
- **Expectation-based**: the total number of rounds required for the protocol to get the system in a legitimate state from an arbitrary state has a constant upper bound [19].

Additionally, weak-stabilizing algorithms exist, which assure stabilization only in the best case.

2.10 The Go Programming Language

2.10.1 Introduction

The Go Programming Language [31], or Golang, is a compiled, statically-typed programming language created at Google and was designed by Robert Griesemer, Rob Pike, and Ken Thompson. The main objective of Go is to increase programming productivity in environments with networked machines and multicore processors. Therefore, it was designed to offer usability and readability of a high-level language, static typing and run-time efficiency of a low-level language, like C, and outrageous performance in multiprocessing and networked processes.

2.10.2 Features

Firstly, Go is an open-source project, thus, everyone can see its implementation, make more efficient functions, or report and fix various bugs. In the language offerings, garbage collection, which is automatic memory management, and memory safety; namely, protection from security vulnerabilities and bugs regarding memory access, such as buffer overflows and dangling pointers, are included. Additionally, CSP-style concurrency is provided, as well as structural programming, in which the equivalence of a type is defined by its actual structure. Anonymous functions and functions inside other functions can be created; hence, many features of functional programming are adopted. Powerful libraries with immerse capabilities are given out of the box, supporting various applications and use cases. Finally, the support for generic programming, i.e., programming style with types to be specified when instantiated, was added in version 1.18.

2.10.3 Concurrency in Go

The CSP-style concurrency provided, implements the formal language of interaction between communicating sequential processes (CSP). Go’s concurrency covers both CPU processes’ parallelism and asynchrony, where the program continues execution despite having some slow operations running. This offers the flexibility and applicability of Go in several environments with different scopes. The main form of its concurrency are light-weight processes, namely, *goroutines*. Goroutines are instantiated by calling a function with the *go* keyword in front. Moreover, a library handling goroutines’ synchronization is provided, with mutex locks and wait group functions, but communication between them is often done by *channels*, which can store messages in FIFO order. Channels are by default send and receive blocking functions unless buffering is used with available space for messages to be exchanged. To avoid blocking operations on channels, the built-in *select* statement can be used.

2.11 The ZeroMQ Messaging Library

2.11.1 Introduction and Features

ZeroMQ [42] (also known as ØMQ, 0MQ or ZMQ) is a high-performance messaging library for asynchronous environments. It is an open-source project based on simplicity and scalability. Its intended use is in both concurrent and distributed systems. Various messages, like serialized and binary data, can be sent over the network by different protocols, such as TCP and UDP, or between processes. Although ZeroMQ provides a message queue for the messages, it does not need an intermediary program responsible for formal messaging protocol translations between senders and receivers, namely message broker [29], to operate. Berkeley sockets, an API for Internet and Unix domain sockets for inter-process communication, is resembled in ZeroMQ's design. Moreover, this API can be used on most operating systems for communication between programs of all kinds.

2.11.2 Socket Types

ZeroMQ provides several sockets in its API for many-to-many communication between endpoints. Each socket is optimized for the particular messaging pattern in which it is intended to be used. Here is a list of the supported sockets [41]:

- **REQ**: Client processes use them to send requests and receive replies from service provider processes. The pattern send, receive, send, receive must be obeyed.
- **REP**: Service provider processes receive requests and send replies to clients. The pattern receive, send, receive, send must be followed.
- **DEALER**: Round-robin algorithms are used for sending and receiving messages from anonymous peers, in reliable fashion – messages do not get dropped. It is an asynchronous replacement for REQ, talking to REP or ROUTER servers.
- **ROUTER**: Explicit addressing is used so that each outgoing message is sent to a specific peer connection. It is REP's equivalent asynchronous socket with the common use case of servers talking to DEALER clients.
- **PUB**: Used by a publisher to distribute data to all connected peers. It is not able to receive any messages.
- **SUB**: Processes subscribe to messages distributed by publishers with this socket. It is not able to send any messages.

- **XPUB**: PUB equivalent, but subscriptions can be received as incoming messages (Subscription message is a byte 1 (for subscriptions) or byte 0 (for unsubscriptions) followed by the subscription body).
- **XSUB**: SUB equivalent, but subscriptions can be created as outgoing messages to the socket (Subscription message is a byte 1 (for subscriptions) or byte 0 (for unsubscriptions) followed by the subscription body).
- **PUSH**: Round-robin algorithms, for sending messages to anonymous PULL peers, are used. Receive operation does not exist.
- **PULL**: Fair-queuing algorithm is used to receive messages from anonymous PUSH peers. Send operation does not exist.
- **PAIR**: A connection to at most one peer can occur. Messages sent over this socket do not get routed or filtered.
- **CLIENT**: Communication between many (1 or more) SERVER peers. Sent messages are scattered among many peers in a round-robin fashion, and messages do not get dropped in normal cases.
- **SERVER**: Communication between many (0 or more) CLIENT peers. Outgoing messages are sent to specific peers. SERVER sockets only provide replies to incoming messages.

2.11.3 Messaging Patterns

The combination of several matching types of the abovementioned socket types enables ZeroMQ to offer many different messaging patterns to exchange messages from senders to receivers. The built-in core patterns are the following [41]:

- **Pub-sub**: a single publisher distributes messages to a set of subscribers (one-to-many communication) - sender does not block, whereas data gets dropped on the receiver side if a hard limit of queued messages is reached.
- **Request-reply**: processes use it to provide or receive services by having asynchronous sockets (DEALER and ROUTER) or synchronous sockets (REQ and REP) (many-to-many communication).
- **Exclusive pair**: connection to only one peer at a time (one-to-one communication), which is usually used for inter-thread messaging when processes are architecturally stable.

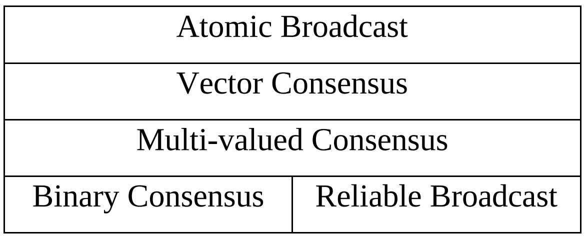
- **Pipeline**: a scalable pattern for task distribution where a few nodes distribute the work to workers and forward the results to a few collectors while no message gets dropped, unless a node disconnects unexpectedly.
- **Client-server**: developed to provide the capability of a server to talk to many clients (many-to-one communication) - clients always initiate the conversation, whereas neither entity drops messages and always blocks if the buffer of the other process is full.

Chapter 3

The Atomic Broadcast Stack and the Studied Self-stabilizing Protocols

3.1 Algorithm’s Structure

The consensus problem is described as the agreement on a certain value in a distributed system, which contains faulty processes. An extension of the consensus problem is atomic broadcast, in which all the correct processes deliver the same values in the same order. As previously mentioned, an atomic broadcast algorithm can be built as a stack of several consensus protocols. This stack is indicated in Figure 3.1 with reliable channels protocols being the foundation of the whole procedure.



Reliable Channels

Figure 3.1: Atomic Broadcast Protocol Stack

Every protocol used in this thesis, to achieve atomic broadcast, is built based on some structural properties. Firstly, all of them are optimally resilient. This means that the stacked consensus algorithms can tolerate up to $f = \lfloor (n - 1)/3 \rfloor$ Byzantine processes out of a total of n in the system. Additionally, they are completely decentralized, without the need of centralized coordination and decisions, and they are time-free. That is, no synchrony assumptions have been made. Performance bottlenecks are avoided by not using digital signatures based on public-key cryptography.

In this thesis, the protocol stack of Correia et al. [15] is proposed as a comparison base between non-self-stabilizing and self-stabilizing consensus algorithms. The studied algorithms that are validated and evaluated for the purposes of this diploma are the Self-stabilizing Vector Consensus and the Self-stabilizing Atomic Broadcast. Both are based on Correia et al. s' versions of VC and ABC. To assess and compare the behavior of self-stabilizing algorithms and their non-self-stabilizing equivalents, the implementation of the remaining stack of the atomic broadcast protocol is the same. That is, for the sake of the comparisons, the same implemented versions of binary consensus and multi-valued consensus, with their use of reliable broadcast, are used for both self-stabilizing and non-self-stabilizing stacks. Therefore, the self-stabilizing stack looks like Figure 3.2.

Self-stabilizing Atomic Broadcast	
Self-stabilizing Vector Consensus	
Multi-valued Consensus	
Binary Consensus	Reliable Broadcast
Reliable Channels	

Figure 3.2: Self-stabilizing Atomic Broadcast Protocol Stack

3.2 Protocols' Stack

The protocol of each part of the stack is discussed in further detail in the following sections. Namely, the atomic broadcast stack of Figure 3.1 is presented, in addition with the two self-stabilizing algorithms that differentiate the two protocols stacks.

3.2.1 Reliable Channels

A distributed system, where messages are delivered if the sender is correct and faults do not occur in transmission, is called *reliable message-passing* [27]. Reliable message-passing environments require the presence of reliable channels between processes [27]. Reliable channels are communication channels between processes in which a fault in message transmission is always detected. Reliable channels are ensured by using message retransmissions and cryptography to correctly identify the sender of the message. Processes share symmetric keys before the execution of the protocols for encrypting and decrypting messages, therefore the decentralized algorithms do not get affected by the key sharing procedure. In our case, ZeroMQ ensures reliable channels between communicating processes because when a message is sent by a correct process it is eventually delivered intact, i.e., without modification, to other correct processes.

3.2.2 Reliable Broadcast

Reliable broadcast is a crucial building block used for every algorithm in the non-self-stabilizing atomic broadcast protocol stack. As explained in Chapter 2, by having RBC in a system several conditions are met. Namely, messages broadcast by correct processes will be eventually delivered, every correct process delivers a message that is delivered by a correct process, and a message in RBC is delivered only once and only if it has already been broadcast.

The RBC in our system is the one proposed by Bracha [7], which consists of four phases, indicated in Figure 3.3.

There are three kinds of messages in this RBC algorithm: *initial, echo,* and *ready*. At first, each process sends its value *v* as an *initial* message to all the processes in the system. Then an *echo* message is sent with the value *v* if and only if the sender has received an *(initial, v)* message or enough supported *echo* or *ready* messages were received, i.e. *(n+f)/2 (echo, v)* or *f+1 (ready, v).* For a ready message to be sent, enough *echo* or *ready* messages must have been received, namely *(n+f)/2 (echo, v)* or *f+1 (ready, v)* respectively. A value *v* gets delivered if and only if *2f+1 ready* messages with value *v* have been received.

ALGORITHM 1. Reliable Broadcast (for process p)

```
step 0: Send (initial, v) to all the processes

step 1: Wait until the receipt of,
        one (initial, v) message or
        (n+f)/2 (echo, v) messages or
        (f+1) (ready, v) messages for some v
        Send (echo, v) to all the processes

step 2: Wait until the receipt of,
        (n+f)/2 (echo, v) messages or
        f+1 (ready, v) messages (including messages received in step 1) for some v
        Send (ready, v) to all the processes

step 3: Wait until the receipt of,
        2f+1 (ready, v) messages (including messages received in step 1 and 2) for some v
        Accept v
```

Figure 3.3 Reliable Broadcast as illustrated in [7, Fig. 1]

3.2.3 Binary Consensus

The binary consensus problem regards the agreement of the correct processes of a system on a binary value $b \in \{0, 1\}$ [7, 33]. The protocol used for this problem is the one in Mostefaoui et al. [33], which satisfies the principles provided in Chapter 2. More precisely, its BC-Validity ensures that a decided value was suggested by a correct process and its BC-Termination and BC-Agreement indicate that every correct process will terminate and agree with each other, with at most one decision i.e., BC-one-shot.

The BC algorithm contains a communication abstraction of a module called binary-value broadcast (BVB) algorithm, shown in Figure 3.4. Basically, when *BV_broadcast* is invoked with a value *v*, *v* gets broadcast. When a process gets *v* from at least one correct process, i.e., at least $f+1$ in total, it then broadcasts *v* to every process. A value $v \in \{0, 1\}$ gets into the *bin_values* set if and only if $2f+1$ instances of *v* have been received.

```
Binary-value Broadcast (for process $p_i$)
operation BV_broadcast MSG($v_i$) is
1: broadcast B_VAL($v_i$).

when B_VAL(v) is received
2: if B_VAL(v) received from (f+1) processes and not yet broadcast: then broadcast
B_VAL(v)
3: if B_VAL(v) received from (2f+1) different processes: then bin_values←bin_values ∪ {v}
```

Figure 3.4 Binary-value Broadcast as illustrated in [33, Fig. 1]

The randomized consensus algorithm, shown in Figure 3.5, has a local variable *est*, which indicates the estimation of the decision. The execution is based on rounds and in each one the BVB gets called with the round number as identifier and the *est* as the value to exchange with others. Then, the process waits for at least one value to be present in *bin_values* to broadcast contained items with the tag *AUX*. The intention behind *AUX* broadcast is to inform the other processes about the estimated values, which were received in BVB by correct processes. Afterwards, the process discards Byzantine-only sent values by waiting for the predicate of line 5 to be true, before entering the local computation phase. The process gets a common coin random value, *s*, and follows one out of the two possible scenarios that depend on the size of the *values* set (defined in line 5). If the *values* set contains both 0 and 1, then the *est* of the next round is the value of the randomly selected *s*. Otherwise, the estimation of the next round is the value *v* contained in the *values* set, but if that value matches the random *s*, the process decides the value *v*.

```
ALGORITHM 2. Binary Consensus protocol (for process $p_i$)
operation propose($v_i$)
$est_i = v_i$
$r_i = 0$
repeat forever
1: $r_i \leftarrow r_i + 1$
2: BVB ($r_i$, $est_i$, EST)
3: wait until (bin_values $\neq \emptyset$)
4: broadcast ($r_i$, w, AUX) where w ∈ bin_values
5: wait until ∃ a set of (n-f) AUX messages delivered from distinct processes such that $values_i$
```

```
⊆ bin_values where $values_i$ is the set of values x carried by these (n-f) messages
6: s ←common_coin()
7: if ($values_i$ = {v}) then
8:        if (v = s) then decide(v) if not yet done
9:        else $est_i$← v
10: else $est_i$← s
end repeat
```

Figure 3.5 Binary Consensus as illustrated in [33, Fig. 2]

3.2.4 Multi-valued Consensus

Multi-valued consensus is defined similarly as binary consensus with the difference that processes have to decide a value from a set of size greater than two [15]. The algorithm proposed can lead processes to decide a proposed value or the default ⊥ value. As a consensus algorithm, agreement and termination as defined in Chapter 2 must be satisfied. The additional principles to be met are covered as three validity propositions in Correia et al. s' work [15]:

- **MVC1 Validity 1**: Same proposal *v* by all correct processes → all correct decide *v*.
- **MVC2 Validity 2**: A decision *v* by a correct process means that *v* is either ⊥ or it was proposed by a process.
- **MVC3 Validity 3**: A value *v* proposed only by faulty processes, is not decided by correct processes.

MVC algorithm, shown in Figure 3.6, uses a vector *V* of size *n*, with each position corresponding to a system process. The function $\#_w(W)$ returns the occurrences of *w* in vector *W* and the received *INIT* messages are saved in *INIT_delivered*. At first, both T1 and T2 tasks are activated concurrently. Task T2 basically receives and stores newly obtained *INIT* messages. Task T1 starts its execution by reliably broadcasting its process' proposal, waits to receive at least *n-f INIT* messages (including its own), and builds a vector *V* with the received messages. Then, if a value *v* exists in *V* at least *n-2f* times (this is also the case when all correct propose the same value), the process broadcasts reliably as a *VECT* message the value *v* with the vector *V*, otherwise the value ⊥ is sent. With the receipt of *n-f VECT* messages (including its own), if no different values were received and a value w appears *n-2f* times, then the process proposes 1 to binary consensus, otherwise 0. According to the binary consensus decision, if the consensus value is 0, then ⊥ is decided, otherwise the process waits to receive enough support from *VECT* messages with value *w*, namely at least *n-2f*, and decides *w*.

ALGORITHM 3. Multi-valued Consensus protocol (for process p_i)

```
Function M_V_Consensus ($v_i$, cid)   {proposal $v_i$, identifier cid}
Initialization:
1: INIT_delivered$_i$ ← $\emptyset$;        {INIT messages delivered}
2: activate task (T1, T2);

Task T1:
3: R_Broadcast ( <INIT, $v_i$, cid, i> );
4: wait until (at least $(n - f)$ INIT messages have been delivered);
5: $\forall_j$ : if (<INIT, $v_j$ , cid, j> has been delivered) then $V_i[j] \leftarrow v_j$ ; else $V_i[j] \leftarrow \perp$;
6: if ($\exists_v : \#_v(V_i) \geq (n - 2f)$) then
7:        $w_i \leftarrow v$;
8: else
9:        $w_i \leftarrow \perp$;
10: R_Broadcast ( <VECT, $w_i$, $V_i$, cid, i> );
11: wait until (at least $(n - f)$ valid messages <VECT, $w_j$ , $V_j$ , cid, j> have been delivered);
12: $\forall_j$ : if (<VECT, $w_j$ , $V_j$ , cid, j> has been delivered) then $W_i[j] \leftarrow w_j$ ; else $W_i[j] \leftarrow \perp$;
13: if ($\forall_{j,k}\, W_i[j] \neq W_i[k] \Rightarrow W_i[j] = \perp$ or $W_i[k] = \perp$) and ($\exists_w$: $\#_w(W_i) \geq (n - 2f)$) then
14:       $b_i \leftarrow 1$;
15: else
16:       $b_i \leftarrow 0$;
17: $c_i \leftarrow$B_Consensus($b_i$, cid);
18: if ($c_i = 0$) then
19:       return $\perp$;
20: wait until (at least $(n - 2f)$ valid messages <VECT, $v_j$ , $V_j$ , cid, j> with $v_j = v$ have been
delivered);
21: return v;
Task T2:
22: when $m_i$ = <INIT, $v_j$ , cid, j> is delivered do
23:       INIT_delivered$_i$ ←INIT_delivered$_i$ $\cup$ {$m_i$};
```

Figure 3.6 Multi-valued Consensus illustrated in [15] as Algorithm 1

3.2.5 Vector Consensus

An agreement on a subset of the proposed values in a distributed system is called vector consensus [15]. This is a useful building block for atomic broadcast only if the majority of the decided values are coming from correct processes. Thus, the values contained in the agreed vector must be at least *2f+1*. The protocol proposed covers the vector validity property, which states that a vector *V* decided by a correct process has at least *f+1* elements from correct processes and the i-th position of the vector contains a proposal of process p_i or $\perp$. The agreement and termination propositions, as indicated in Chapter 2, are also satisfied.
The algorithm, shown in Figure 3.7, works in rounds. After reliably broadcasting its proposal, each process starts its execution from round zero (*r=0*). In each round, a process waits to receive cumulatively from the beginning of the execution *n-f+r* messages from RBC. Then, a vector *W* is build based on the received values. Position j contains $\perp$ if no message was received from p_j, otherwise its value is placed in cell j. The vector is proposed to multi-valued consensus. If the consensus of MVC is not $\perp$, then that vector is the decision of VC, otherwise the process continues with the execution of the next round.

ALGORITHM 4. Vector Consensus protocol (for process p_i)

```
Function Vector_Consensus ($v_i$, vcid)          {proposal $v_i$, identifier vcid}
1: $r_i \leftarrow 0$;        {round number}
2: R_Broadcast ( <VC_INIT, $v_i$, vcid, i> );
3: repeat
4:        wait until (at least($n-f+r_i$) VC_INIT messages have been delivered);
5:        $\forall_j$ : if ( <VC_INIT, $v_j$ , vcid, j> has been delivered) then $W_i[j] \leftarrow v_j$ ; else $W_i[j] \leftarrow \perp$;
6:        $V_i \leftarrow$ M_V_Consensus ($W_i$, (vcid,$r_i$));
7:        $r_i \leftarrow r_i + 1$;
8: until ($V_i \neq \perp$);
9: return $V_i$;
```

Figure 3.7 Vector Consensus illustrated in [15] as Algorithm 2

3.2.6 Atomic Broadcast

As mentioned before, total reliable broadcast is the problem of delivering the same messages in the same order to every process. The algorithm proposed by Correia et al. [15], which is shown in Figure 3.8, uses vector consensus as a building block. Moreover, it satisfies each one of the four atomic broadcast properties mentioned in Chapter 2, which indicates the correctness of the protocol.

At first, an initialization phase exists where tasks T1 and T2 are activated to run concurrently, and the message number, *num*, and ABC identifier are set to zero. Broadcasting atomically a value requires from a process to call the *A_broadcast* procedure. Inside that function, the value is reliably broadcast with the current message number, *num*, and after that, *num* gets incremented by 1. The uniqueness of each message is guaranteed by the uniqueness of *num*'s value. Task T2, which does similar work with Task T2 of MVC, gets messages from RBC module and inserts them in the *R_delivered* set. Task T1, however, does the whole work of the protocol. The process tries to decide, with other processes, the order to deliver messages in *R_delivered*, whenever *R_delivered* contains a proposal. In the beginning, a vector *H* with the hashes of the messages in *R_delivered* is created. Then, *H* is proposed to the VC module, which basically decides a vector *X*. Each cell j of the vector *X* contains either the vector *H* of the process j or ⊥. Vector *X* has at least *2f+1 H* vectors from different processors. A message gets into *A_deliver* if the process is confident that it has been proposed by at least one correct process. To ensure this, the hash of that message appears in at least *f+1 H* vectors inside the decided *X* vector. The process waits to receive in *R_delivered* every message that is in *A_deliver*. Then it delivers each message in a pre-determined deterministic order and removes them from *R_delivered*.

ALGORITHM 5. Atomic Broadcast protocol (for process p_i)

```
Initialization:
1: $R\_delivered_i \leftarrow \emptyset$; {messages delivered by the reliable broadcast protocol}
2: $aid_i \leftarrow 0$; {atomic broadcast identifier}
3: $num_i \leftarrow 0$; {message number}
4: activate task (T1, T2);

When Procedure A_Broadcast (m) is called do
5: R_Broadcast ( <A_MSG, $num_i$, m, i> );
6: $num_i \leftarrow num_i + 1$;

Task T1:
7: when ($R\_delivered_i \neq \emptyset$) do
8:     $H_i \leftarrow$ {hashes of the messages in $R\_delivered_i$};
```

```
9:      $X_i$ ←Vector_Consensus ($H_i$, $aid_i$);
10:     wait until (all messages with hash in f+1 or more cells in vector $X_i$ are in
        $R\_delivered_i$);
11:     $A\_deliver_i$ ←{all messages with hash in f+1 or more cells in vector $X_i$};
12:     atomically deliver messages in $A\_deliver_i$ in a deterministic order;
13:     $R\_delivered_i$ ← $R\_delivered_i$ - $A\_deliver_i$;
14:     $aid_i$ ← $aid_i$ + 1;

Task T2:
15: when <A_MSG, num, m, i> is delivered by the reliable broadcast protocol do
16: $R\_delivered_i$ ←$R\_delivered_i$ ∪ {<A_MSG, num, m, i>};
```

Figure 3.8 Atomic Broadcast illustrated in [15] as Algorithm 3

3.3 Self-stabilizing Protocols

Self-stabilization provides greater safety coverage as a stronger model of failure tolerance. Transient faults can alter the memory of a process, resulting inconsistent state of the program, or create transmission errors, such as corruption or reordering [37]. In this distributed environment, the system could be initialized inconsistently, or its state could not be compatible with the rest of the program, for example after recovery from a process failure [37]. Therefore, to overcome these faults with the existence of Byzantine processes and bring the system back to a safe state many things must be considered. One thing to be mentioned is that despite the changes of a system's state, the program itself is inviolable, i.e., its behavior does not change. Moreover, in our system, the processes have an ID value in the range of *0* to *n-1*.

The newly developed algorithms, namely Self-stabilizing Vector Consensus and Self-stabilizing Atomic Broadcast are based on the non-self-stabilizing protocols of VC and ABC, which were developed by Correia et al. [15] and are described above. These algorithms provide optimal resilience as a base for consensus and their self-stabilizing equivalents have to satisfy additional properties for transient handling capabilities. To begin with, a wait-free operation is mandatory for a self-stabilizing solution to be valid. The reason behind this is the fact that the program counter can be corrupted and point to any part of the program. That is, if the program counter points to a wait command, for one or several processes, but did not perform other necessary actions, as indicated by the normal execution, the processes could be deadlocked in that portion of the code. Thus, to avoid indefinite halt by program counter fault, wait-until statements should be transformed to if-statements, while a unification of every message exchanged should take place. In addition, the program could be transferred in an arbitrary state, even at the beginning of its execution, thus a repetitive behavior is obligatory for correction of itself. This implies the use of a *do-forever* loop for the program's structure, namely for checks and instructions to be executed indefinitely until the protocol satisfies all its properties and delivers a value.

At first, a self-stabilizing version of the reliable broadcast is embedded inside both algorithms with every message sent being stored. The Byzantine Reliable Broadcast used for self-stabilizing transformation is the one created by Bracha [7], described in Section 3.2.2. In Bracha's RB, there are three kinds of messages; *init*, *echo*, *ready* and all of them must be saved in the self-stabilizing version. Therefore, an array *msg* is utilized, which stores the messages of a process *i* in *msg[i]* with each RB type (*rbMSG* $\in$ *{init, echo, ready}*) message stored in *msg[i][rbMSG]*. The unification of the messages for wait-free execution is provided in the fact that each process broadcasts all its *init*, *echo,* and *ready* messages every time. Packet loss and the altering of messages saved due to transient faults are tolerated by broadcast repetition and additional checks. The duplication of messages is not a problem as a virtue of the storage of the received messages as sets by the unification operation. Every wait operation of the original RB is transformed to an if-statement which performs checks on the content of the *msg* array. The reliably delivered values are calculated repetitively in each iteration and are used with the appropriate actions in the new protocols.

3.3.1 Self-stabilizing Vector Consensus

In this section, details for the SSVC algorithm are provided. The problem to be solved is defined in Section 3.2.5 as the agreement on a subset of the proposed values in a distributed system. The pseudo-code of the algorithm is shown in Figure 3.9. The *getValue()* interface provides the proposal given to the protocol by the upper layer. The only variable utilized for the algorithm is the *msg* array, which usage is described previously. This array stores messages in the corresponding *msg[process_id][rbMSG]* as sets, i.e., no duplicates exist. The messages stored are composed as *(k,m)* tuples, where *k* is the sender's ID and *m* is the value proposed by *k*.

The message receival part of the protocol handles new messages with some inspections. When a new message from p_j arrives, the new *echo* and *ready* messages are saved in *msg[j]*. The newly received *init* message is stored in *msg[j]* if there is not a new *echo* or *ready* message *(k,m')* received, while p_i knows that p_j has received *(k,m)* from some p_k and $m \neq m'$.

Additionally, there are some functions used in the main body of SSVC. *RBcast()* just inserts the proposal, *getValue()*, in process suggestion, *msg[i][init]*. The *RB_Deliver(k)* function takes as input the ID of a process. A value *m* is returned by it if at least *2f+1* nodes have sent a *(k,m) ready* message to the caller process, otherwise ⊥ is returned. To evaluate this, the examination that takes place is whether *(k,m)* exists in *msg[b][ready]* of *2f+1 b* values, i.e., processes. Finally, the *result()* function returns a value based on three different scenarios:

- If MVC module has returned ⊥ or the *msg[i][init]* is empty, then ⊥ is returned.
- If MVC module has returned a transient fault, then the default transient symbol Ψ is returned.
- If MVC module has returned a value, then that value is returned to the upper layer.

Inside the main body of the protocol, namely the *do-forever* loop, several checks are made to revoke transient faults or to reliably deliver values. *Msg[i]* indicates the messages sent by process i, p_i, whom code is shown below. At first, the process checks whether an *echo* message was sent by process p_j, but an *init* message was not. Also, p_i scans *ready* messages to find if one exists that is not reported by either *(n+f)/2* processors who have seen an *echo* of this or by *f+1* who have seen a *ready* of this. If either proposition is true, that means that a transient fault has occurred, and each type of message set is cleared. After the initial check, *RBcast()* is called and a loop over every process p_k of the system begins. If either p_i has more than one *init* value from p_k or p_i holds an *init* message in *msg[k][init]* that is neither ∅ nor some message from p_k or p_i holds two conflicting *echo* or *ready* messages in *msg[k]* then p_k's entry is reset to ∅. Then, an *echo* message *(k,m)* is added in p_i's *msg* entry when an *init* message *(k,m)* from p_k is found and it is not already in *msg[i][echo]*. Furthermore, a *ready* message tuple *(k,m)* is appended to p_i's *msg* if *(k,m)* is received by either *(n+f)/2* processes as an *echo* message or by at least *f+1* nodes as a *ready* message. When exiting the processes' loop, process p_i performs a broadcast of its *msg[i]* values. Finally, a vector of size *n*, which has a position for each process ID, is created by invoking *RB_Deliver()* with argument j, for each position j. The MVC module gets invoked with the vector transformed as a value proposal, as long as the vector itself has at least *n-f* not ⊥ entries. If the vector does not pass this check, then the algorithm continues from the beginning of the *do-forever* loop. Otherwise, if the MVC module had been called and had provided a consensus value, then the *result()* function is called. As it is shown below, the time complexity of a *do-forever* iteration is $O(n^3)$, where *n* is the number of processes of the system, due to the steps needed to check every message when *msg* has the maximum number of values proposed. Namely, when each message type of every process' entry contains one message for each process, i.e., *msg* includes *O(n^2)* values, thus, to evaluate message occurrence *O(n)* times in processes' loop (line 9), *O(n^3)* steps are required.

ALGORITHM 6. Self-stabilizing Vector Consensus protocol (for process p_i)

Interfaces:

getValue(): returns the value v from the calling function (upper layer).

Variables:

Array msg[x][rbMSG] is an array of messages with the reliable broadcast message field type rbMSG ∈ {init, echo, ready}. Array msg_i[i][typ] stores p_i's messages of rbMSG type typ.

Operations:

```
/* Invocation operation: Reassign the value v = getvalue() to msg[i][init] */
RBcast():
1:      msg[i][init] ← msg[i][init] ∪ getValue()
/* If there exists a ready message m supported by 2f + 1 then deliver */
RB Deliver(k):
2:      if $\exists m : (2f + 1) \le |\{p_\ell \in P(k, m) \in msg[p_\ell][ready]\}|$ then return m else return ⊥;

/* Returns results in {⊥, Ψ} ∪ V where V is the domain of values that may be proposed by
processors via multivalued consensus */
result() :
3:      if MVC.result() = ⊥ ∨ msg[i][init] = ∅ then return ⊥;
4:      if MVC.result() ∉ {⊥} ∪ V then return Ψ;
5:      if MVC.result() = m then return m.v;

do forever begin
        // Consistency checks. Resets to empty set
6:      if $\exists(j,m) \in msg[i][echo]\ (m \notin msg[j][init]) \vee \exists(j,m) \in msg[i][ready]\ \neg((n + f)/2 <$
$|\{p_\ell \in P : (j, m) \in msg[\ell][echo]\}| \vee (f + 1) \le |\{p_\ell \in P : (j, m) \in msg[\ell][ready]\}|)$ then
7:              foreach s ∈ rbMSG do msg[i][s] ← ∅;
8:      RBcast();
        // Broadcast phase transition
9:      foreach $p_k \in P$ do {
```

```
10:             if $|msg[k][init]| > 1 \lor msg[k][init] \notin \{\emptyset, \{(k, -)\}\} \lor \exists s{\neq}init \exists p_j{\in}P$
$\exists (j,m),(j,m'){\in}msg[k][s]\ m \neq m'$ then
11:                 $msg[k][s] \leftarrow \emptyset$;

12:             if $\exists m \in msg[k][init]$ then $msg[i][echo] \leftarrow msg[i][echo] \cup \{(k, m)\}$;

                // add new ready messages if conditions are met
13:             if $\exists m\ (n + f)/2 < |\{p_\ell \in P : (k, m), msg[\ell][echo]\}|$ then $msg[i][ready] \leftarrow$
$msg[i][ready] \cup \{(k, m)\}$;
14:             if $\exists m\ (f + 1) \leq |\{p_\ell \in P : (k, m) \in msg[\ell][ready]\}|$ then $msg[i][ready] \leftarrow$
$msg[i][ready] \cup \{(k, m)\}$;
            }
            // Send
15:     broadcast MSG(msg[i]);
        // Compile a vector of values that are RB Delivered
16:     foreach $p_k \in P$ do $V[k]$ := RB Deliver(k);
        // Check if ready to call MV consensus
17:     if MVC.result() = $\perp \land (|\{p_x \in P : V[x] \neq \perp\}| \geq n - f)$ then MVC.propose(V);

// Receive actions
18: upon MSG(mJ) arrival from $p_j$ do foreach $s \in rbMSG, p_k \in P : s \neq init \lor$
$\nexists s{\neq}init,(k,m),(k,m'){\in}(msg[j][s]{\cup}mJ[s])\ m \neq m'$ do $msg[j][s] \leftarrow msg[j][s] \cup mJ[s]$;
```

Figure 3.9 Self-stabilizing Vector Consensus

3.3.2 Self-stabilizing Atomic Broadcast

The problem to be solved is the total broadcast, mentioned in Chapter 2. The self-stabilizing atomic broadcast is an extension of the ABC of Correia et al. [15], which is described in Section 3.2.6. The development of this algorithm is similar to the SSVC with several actions shared by both protocols. The pseudo-code of the algorithm is presented in Figure 3.10. SSABC uses a bounded counter *num,* in the range of [0, MAXINT], to locally order messages along with the *msg* array used in SSVC. The *getValue()* interface returns the next client request which is not already atomically delivered. The *RB_delivered* and *AB_Delivered* sets mentioned below correspond to reliably delivered tuples and atomically delivered tuples, respectively. The messages stored in *msg* are composed as *(num,m,k)* tuples, where *num* is the value of the local counter *num* of the process p_k that received the request *m*, *k* is sender's ID and *m* is the value proposed by *k*. The *num* part of the message tuple has a critical role in the behavior of the algorithm. It helps servers recognize that a newly received request, which is identical to a previous one, is indeed a distinct one which has not been already delivered so they try to atomically broadcast it. An important design decision was the handling of one client request per time. That is, servers only have to manage to deliver one request and then move on to the next one. This makes the protocol invulnerable to Denial-of-Service Attacks as no malicious process can monopolize server's attention.

As with the SSVC module, SSABC has some specific receive actions and some clearly defined functions that are being used. Its receival behavior is identical with that of its beneath stack protocol. However, some differences exist in the definitions of some functions. *RBcast()* puts the current request to be delivered in *msg[i][init]* as a tuple, but when a new request is ought to be handled, that is after the delivery of the current one, the new request is inserted as a newly constructed tuple before increasing *num* by one. A new method *AB_Deliver()* is responsible for atomically delivering values. It gets the decided SSVC vector as input, which is basically a vector of vectors. That is, the vector is composed of one vector per process, which can be either ⊥ or a proposal from a process to the SSVC. The function delivers in a pre-determined deterministic order every value *m* that was reliably delivered, i.e., present in *RB_Deliver*, and exists in at least *f+1* vectors of the SSVC decision. The values that are atomically delivered are also returned as a set of tuples. The *RB_Deliver(k)* and *result()* functions are like the ones used in SSVC, with some small changes in the *result()* one. Instead of regarding the possible values of the MVC module, it takes into consideration the decision of SSVC. If transient Ψ or ⊥ is returned, then no action is taken, otherwise the value *true* is returned.

The *do-forever* loop, in which several assessments are made to deal with transient faults or to reliably deliver values, is the core execution part of the protocol. As with SSVC, *msg[i]* indicates the messages sent by process i, p_i, whom code is shown below. At first, the process makes the same checks as the first step of SSVC in addition with some extras that include the *num* counter. Namely, evaluations on whether *num* is larger or equal to its maximum value, a message tuple from p_i has *num* larger or equal to the current *num's* value, or an *echo* message was sent by process p_j, but an *init* message was not, take place. Also, p_i assesses if a *ready* message exists, and it is not reported by either *(n+f)/2* processors who have seen an *echo* of this or by *f+1* who have seen a *ready* of this. If either proposition of the first step evaluations is true, that means a transient fault has occurred. This results the clearance of each type of message set, while *num* is set to 0. The following checks do not consider the *num* section of a process message as it is not required. The reason behind this is that at most one request is proposed by every process so a distinction between two values from the same process is not needed. Thus, the assessments of the remaining code check only the *(k,m)* part of the tuple, as shown in the pseudo-code from line 17, for clearly reading purposes. In reality the whole 3-tuple is stored as a message. After the initial check, *RBcast()* is called and a loop over every process p_k of the system begins. If either p_i has more than one *init* value from p_k or p_i holds an *init* message in *msg[k][init]* that is neither ∅ nor some message from p_k or p_i holds two conflicting *echo* or *ready* messages in *msg[k]* then p_k's corresponding entry is reset to ∅. Then, an *echo* tuple *(k,m)* is added in p_i's *msg* when an *init* message *(k,m)* from p_k is found that is not already in *msg[i][echo]*. Furthermore, a *ready* message *(k,m)* is appended to p_i's *msg* if *(k,m)* was sent to p_i by either $(n+f)/2$ processes as an *echo* message or by at least $f+1$ nodes as a *ready* message. When exiting the processes' loop, process p_i performs a broadcast of its message tuples, *msg[i]*. Afterwards, the reliably delivered but not atomically delivered messages are calculated from p_i's *msg*. That is, every *ready* tuple, which is sent by at least $2f+1$ different processes, is contained in set *RB_delivered*. Finally, if there is not a SSVC instance running and *RB_delivered* is not empty, then *RB_delivered* tuples are proposed as one value in SSVC. When this protocol provides a decision, the *result()* function is called. If the *result()* returns true, i.e., the decision is a valid vector, *AB_deliver()* method is called, otherwise no action is taken. The returned *AB_delivered* values are removed from the reliably delivered ones and an *ACK* message is replied to the client that provided the request. The server is ready to take a new request, unless the atomically delivered messages do not include its current one. In any case, the algorithm continues its execution to be available to handle new incoming queries.

As it is indicated below, the time complexity of a *do-forever* loop is $O(n^3)$, where n is the number of processes in the system, due to the steps needed to count every message when *msg* has the maximum number of values proposed. Namely, when each message type of every process' entry contains one message for each process, the *msg* includes $O(n^2)$ values, thus, to evaluate message occurrence $O(n)$ times in processes' loop (line 16), $O(n^3)$ steps are required.

ALGORITHM 7. Self-stabilizing Atomic Broadcast protocol (for process p_i)

Interfaces:

getValue(): returns the pending request, that was not already atomically delivered.

VC is a vector consensus instance. The algorithm considers at most one such instance at any time. VC.propose(V) initiates the instance of VC and proposes an ordered set V. When there is no instance of VC initiated then VC = ⊥.

Variables:

Array msg[x][rbMSG] is an array of messages with the reliable broadcast message field type rbMSG $\in$ {init, echo, ready}. Array msg_i[i][typ] stores p_i's messages of rbMSG type typ.
Set RB_delivered maintains the messages that have been reliably delivered but have yet to be atomically delivered.
Counter num_i is the message number in [0, MAXINT], used by p_i to locally order the messages it proposes.

Operations:

```
RBcast(): do
1:          if NEW REQUEST RECEIVED $\vee$ $num_i = 0$ then
2:               {msg[i][init] $\leftarrow$ $\langle num_i$, getValue(), i$\rangle$; $num_i \leftarrow num_i + 1$};
3:          else
4:               msg[i][init] $\leftarrow$ msg[i][init] $\cup$ $\langle$max(0, $num_i$-1), getValue(), i$\rangle$;

// Return the ordered set of messages that exist in f + 1 places in the agreed vector
AB_Deliver(V): do
5:     foreach $p_i \in P$ and $m \in V$ [i] do:
            if m exists in f + 1 vectors V [k] then add m to set AB_delivered;
6:     AB_delivered.deliver() ; // Deliver messages in deterministic order.
7:     return AB_delivered;

/* If there exists a ready message m supported by 2f + 1 then deliver */
8: RB_Deliver(k) do if $\exists m : (2f + 1) \le |\{p_\ell \in P(k, m) \in msg[p_\ell][ready]\}|$ then return m else
return $\perp$;

result() :
9:     if VC.result() = $\perp$ $\vee$ msg[i][init] = $\emptyset$ then return $\perp$;
10:    if VC.result() = $\Psi$ then return $\Psi$;
11:    if VC.result = V then return True;
```

do forever begin

```
       // Consistency checks. Resets to empty set
12:    if ($num_i \geq$ MAXINT) $\vee$ $\exists$(num,m,i)$\in$msg[i] num $\geq num_i$ $\vee$ $\exists$(j,m)$\in$msg[i][echo] m $\notin$
```

```
msg[j][init] $\lor$ $\exists$(j,m)$\in$msg[i][ready] $\neg$ ((n + f)/2 < |{$p_\ell$ $\in$ P : (j, m) $\in$ msg[$\ell$][echo]}| $\lor$ (f + 1)
$\leq$ |{$p_\ell$ $\in$ P : (j, m) $\in$ msg[$\ell$][ready]}|) then
13:             foreach s $\in$ rbMSG do msg[i][s] $\leftarrow$ $\emptyset$;
14:             $num_i$ $\leftarrow$ 0;

15:     RBcast();

        // Broadcast phase transition
16:     foreach $p_k$ $\in$ P do
17:             if |msg[k][init]| > 1 $\lor$ msg[k][init] $\notin$ {$\emptyset$, {(k, $-$)}} $\lor$ $\exists$s$\neq$init$\exists p_j \in$P
$\exists$(j,m),(j,m′)$\in$msg[k][s] m $\neq$ m′ then
18:                     msg[k][s] $\leftarrow$ $\emptyset$;
19:             if $\exists$m $\in$ msg[k][init] then msg[i][echo] $\leftarrow$ msg[i][echo] $\cup$ {(k, m)};

20:             if $\exists$m(n + f)/2 < |{$p_\ell$ $\in$ P : (k, m), msg[$\ell$][echo]}| then msg[i][ready] $\leftarrow$
msg[i][ready] $\cup$ {(k, m)};
21:             if $\exists$m(f + 1) $\leq$ |{$p_\ell$ $\in$ P : (k, m) $\in$ msg[$\ell$][ready]}| then msg[i][ready] $\leftarrow$
msg[i][ready] $\cup$ {(k, m)};
        // Send
22:     broadcast MSG(msg[i]);
23:     RB_delivered $\leftarrow$ RB_delivered $\cup$ RB_Deliver(i);
        // If VC complete then deliver VC agreed messages and remove these messages from
RB delivered set
24:     if VC.result() = V then RB_delivered $\leftarrow$ RB_delivered \ AB_Deliver(V);
        // Call an instance of vector consensus
25:     if VC.result() = $\perp$ $\land$ RB_delivered $\neq$ $\emptyset$ then VC.propose(RB_delivered);

// Receive actions
26: upon MSG(mJ) arrival from $p_j$ do foreach s $\in$ rbMSG, $p_k$ $\in$ P : s $\neq$ init $\lor$
$\nexists$s$\neq$init,(k,m),(k,m′)$\in$(msg[j][s]$\cup$mJ[s]) m $\neq$ m′ do msg[j][s] $\leftarrow$ msg[j][s] $\cup$ mJ[s];
```

Figure 3.10 Self-stabilizing Atomic Broadcast

Chapter 4

Implementation

4.1 Implementation Decisions

The first decisions that must be taken for a randomized BFT algorithm development in an asynchronous environment is the programming language and the communication library to be used. As mentioned earlier, the Go programming language was used for the implementation and the ZeroMQ library for the asynchronous message exchanges.

Go is the de facto programming language for concurrent and parallel development [39]. In our case, an asynchronous environment implies the usage of multiple threads locally to handle communication and for the logic of the program. Therefore Go, by simplifying concurrent programming with the use of *goroutines* and *channels* to handle their communication, is a perfect fit for the case of the atomic broadcast development. On top of that, Go is fast, with its direct compilation to machine code, and scales well as each *goroutine* takes up only 2kB of memory, which is ideal for the experimental evaluation of our system. The community of developers has grown a lot throughout the years because of its open-

source nature. The provided resources regarding Go, either official documentation or developers' feedback, helped tremendously the implementation of the algorithms.
Furthermore, the general philosophy of ZeroMQ is to adhere on the technical goals of scalability and simplicity. With the open-source form of the protocols developed, ZeroMQ is the state-of-the-art library for distributed environments. Its widely spread nature flows from the fact that it is fast, lightweight and assures message delivery by using hidden queues, which provide reliability. Moreover, the implementation of the non-self-stabilizing atomic broadcast stack was done with ZeroMQ as the message-passage medium so for more accurate comparisons with the studied algorithms its usage is obligatory.

4.2 Project Structure

The general project structure followed is based on the implementation of the non-self-stabilizing atomic broadcast stack, which was done in a previous thesis [35]. Primarily, the usage of that project structure is for compatibility with that implementation in order to be able to execute both atomic broadcast algorithms and to compare them in a similar environment. That is, an identical client behavior was used to provide accuracy to the studied protocols' comparisons. Additionally, this compatibility was done to match and connect the studied algorithms with the implementation of the previous protocol stack, i.e., self-stabilizing vector consensus connection with the multi-valued consensus. The full working code of the project can be found on my GitHub repository [4].
To begin with a brief description of the structure, a different Go project was used for servers' actions and clients' behavior. The two project structures are indicated in Figures 4.2 and 4.10 for servers and clients respectively. Servers and clients share a similar structure with several common packages. Information and error messages of the execution of both are written in log files by the logger implemented in *logger* folder. The configuration of the appropriate scenario to be run with the set-up of the required IP addresses or ports of the system is handled in the *config* folder. The *variables* file contains shared variables of the program, like servers and clients number, maximum possibly existing Byzantine processes, and the ID of each process. The message structures, for each protocol, are held in the *types* package for processes to have a well-defined form of communication. The *messenger* package handles the message exchanges between either servers or servers and clients by using the ZeroMQ. The only thing that differentiates servers from clients is their general behavior as every algorithm mentioned in Chapter 3 is stored in the servers' *module* package, whereas clients' actions are contained in the *app* package.

4.3 Communication

The communication structure used is the same as the one used in a previous thesis [35] as a virtue of developing the self-stabilizing algorithms in the same system. Therefore, the communication decisions made in that previous work are inherited in this project. As mentioned earlier, *messenger* makes use of ZeroMQ library for communication between processes. Each server deploys different kind of sockets, namely a REQ/REP pair for each server to communicate (request/reply) with each other and for each client a REP socket to get requests and a PUB one to send a response. On the other side, clients make use of a REQ and a SUB socket for every server, to send requests and receive replies respectively. The general socket formation is shown in Figure 4.1.

When a server receives a request, it immediately responds with an empty message to make the REP socket free for the next request. Therefore, this avoids the duplicate consecutive sends and receives, which would crash the system. Neither clients nor servers block waiting for a message from the other side. Moreover, the preference for the usage of REQ/REP sockets is based on their reliability, despite being synchronous. To circumvent the synchrony of these sockets, a timeout and retransmission of messages was embedded in the project's implementation with the use of Go. The CLIENT/SERVER sockets were added to ZeroMQ after the completion of the older thesis [35] so they were not utilized.

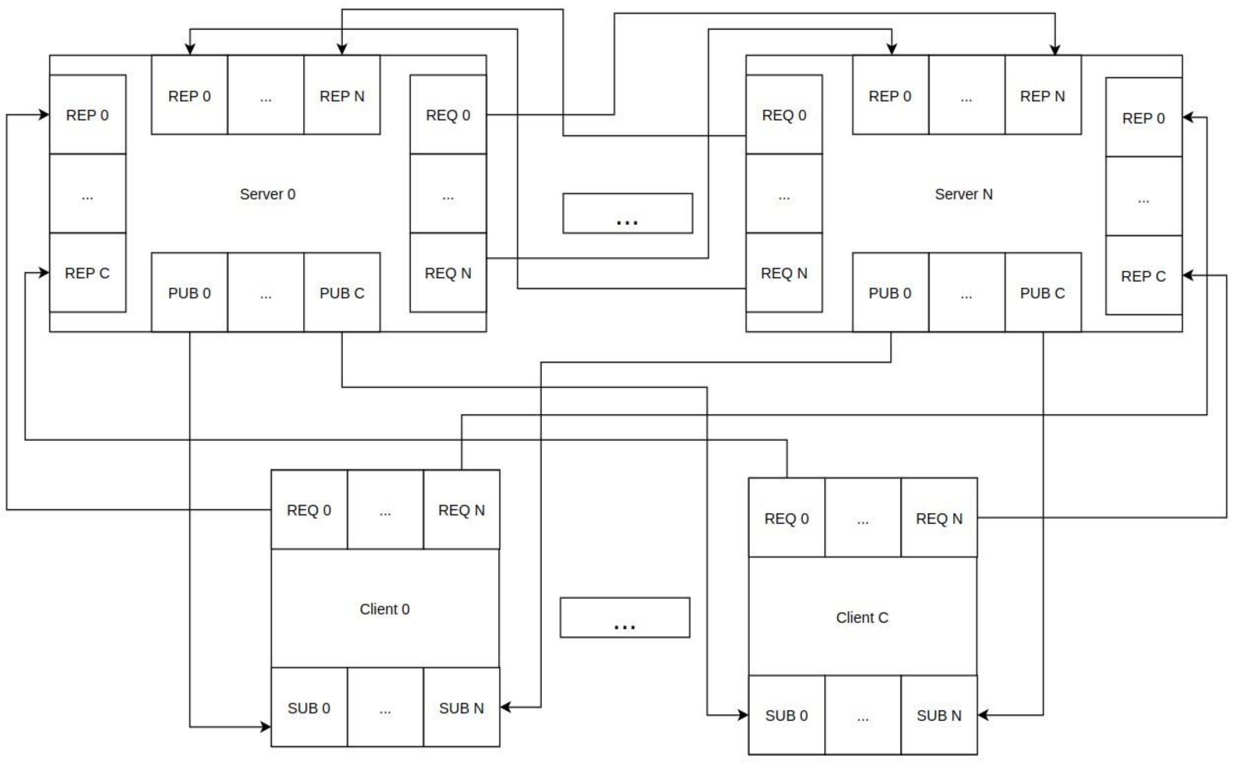


Figure 4.1 Communication organization with ZeroMQ sockets

4.4 Server Implementation

A closer look to the server project structure is indicated in Figure 4.2. Servers communicate both with other servers, to achieve consensus, and with clients, to exchange requests and replies. Every algorithm mentioned in Chapter 3 is implemented in the *modules* package with the intention of atomically broadcasting a message. Each server has a character array, in which clients' request are delivered. The main objective is to have every server agree on the sequence of the delivered characters, thus achieving total order of messages.
Sockets and *goroutines* for communication are the first things to be initiated with the launch of a server. After that, global variables of the program, like servers and clients number, the quantity of Byzantine processes, server's ID, and the scenario to be executed are set to the appropriate values. The atomic broadcast or its self-stabilizing counterpart module is initialized, based on the execution scenario, while Request Handler gets ready to handle clients' messages. Each server runs indefinitely or until it is shut down.

BFTWithoutSignatures

- config
    - ip.go
    - local.go
    - scenario.go
- faults
    - byzantine.go
    - transient.go
- logger
    - logger.go
- messenger
    - messenger.go
- modules
    - self_stabilized_atomic_broadcast.go
    - self_stabilized_vector_consensus.go
    - request_handler.go
    - atomic_broadcast.go
    - vector_consensus.go
    - multivalued_consensus.go
    - binary_consensus.go

    - reliable_broadcast.go
- threshenc
    - key_generator.go
    - key_reader.go
    - sign_and_verify.go
- types
    - message.go
    - abc_ message.go
    - vc_ message.go
    - mvc_ message.go
    - bc_ message.go
    - rb_ message.go
    - client_ message.go
    - reply_ message.go
    - ssabc_ message.go
    - ssvc_ message.go
- main.go

Figure 4.2 Server project structure

4.4.1 Config

This package contains the information regarding the scenario to be executed and the low-level communication details between processes.

In the *scenario.go* file, a variable indicates the existence of transient faults in the to be executed test and defines the behavior of Byzantine processes i.e. idle, sending different or the same messages to processes.

For process communication, the *local.go* or the *ip.go* programs are called for local or real-world executions, respectively. In *local.go*, a different port number is given to each process, which matches the corresponding socket type, whereas in the *ip.go* file, the IP address of each computer is also present. Figure 4.3 indicates port assignment for processes in the localhost scenario.

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

Figure 4.3 Ports configuration for local execution as illustrated in [35, Fig. 4.4]

4.4.2 Types and Variables

Every message sent in any part of the protocol stack consists of multiple fields, thus, message types are built in the system as Go *structs*. On the other hand, messages are sent as bytes through ZeroMQ, which implies the necessity of serialization and deserialization functions for our types. This obligation is fulfilled with a package called Gob [24], which manages stream of bytes exchanged between processes. Despite the encoding support of Go's primitive types, the encoding of a more complex structure, like the message types created for our system, requires the *GobEncoder* and *GobDecoder* functions to be implemented as an interface for byte transformation.

A general structure of a message used is define in the *message.go* file. This is a wrap up message for all different types, which includes the payload as a byte of streams, the type of the underlying message, and its origin. This structure is encoded as bytes to be sent over the network. This provides a more uniform method of communication which hides extra complexities and delays of handling different types of messages. According to the type of the message, its payload is decoded as the corresponding message type *struct* and it is forwarded to the appropriate channel to be handled by the process. The implementation of the studied self-stabilizing vector consensus and atomic broadcast types, as well as the simple atomic broadcast one, are shown in Figures 4.4, 4.5, and 4.6, respectively. In their *type* folder, both self-stabilizing protocols contain the implementation of a tuple of the suggestion exchanged in their build in reliable broadcast module as well as the message sent to other processes.

```
// Tuple containing the sender and the corresponding value of the sender
type SSVCMessageTuple struct {
	Sender int
	Value []byte
}

// self stabilizing vector consensus SSVCMessage - SSvector consensus message struct
type SSVCMessage struct {
	SSVCid int
	Content map[string][]SSVCMessageTuple
}

// NewSSVCMessage - Creates a new self stabilizing VC message
func NewSSVCMessage(id int, content map[string][]SSVCMessageTuple) SSVCMessage {
	return SSVCMessage{SSVCid: id, Content: content}
}

// GobEncode - vector consensus message encoder
func (ssvcm SSVCMessage) GobEncode() ([]byte, error) {
	w := new(bytes.Buffer)
	encoder := gob.NewEncoder(w)
	err := encoder.Encode(ssvcm.SSVCid)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	err = encoder.Encode(ssvcm.Content)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return w.Bytes(), nil
}

// GobDecode - vector consensus message decoder
func (ssvcm *SSVCMessage) GobDecode(buf []byte) error {
	r := bytes.NewBuffer(buf)
	decoder := gob.NewDecoder(r)
	err := decoder.Decode(&ssvcm.SSVCid)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	err = decoder.Decode(&ssvcm.Content)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return nil
}
```

Figure 4.4 Self-stabilizing Vector Consensus message type [4]

```
// Tuple containing the sender and the corresponding value of the sender
type SSABCMessageTuple struct {
	Sender int
	Num    uint32
	Value []byte
}

// self stabilizing atomic broadcast - SSABCMessage message struct
type SSABCMessage struct {
	Content map[string][]SSABCMessageTuple
}

// NewSSABCMessage - Creates a new SS ABC message
func NewSSABCMessage(content map[string][]SSABCMessageTuple) SSABCMessage {
	return SSABCMessage{Content: content}
}

// GobEncode - SS atomic broadcast message encoder
func (abcm SSABCMessage) GobEncode() ([]byte, error) {
	w := new(bytes.Buffer)
	encoder := gob.NewEncoder(w)
	err := encoder.Encode(abcm.Content)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return w.Bytes(), nil
}

// GobDecode - SS atomic broadcast message decoder
func (abcm *SSABCMessage) GobDecode(buf []byte) error {
	r := bytes.NewBuffer(buf)
	decoder := gob.NewDecoder(r)
	err := decoder.Decode(&abcm.Content)
	if err != nil {
		logger.ErrLogger.Fatal(err)
	}
	return nil
}
```

Figure 4.5 Self-stabilizing Atomic Broadcast message type [4]

```
// AbcMessage - atomic broadcast message struct
type AbcMessage struct {
	Num   int
	Value []byte }

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

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

Figure 4.6 Atomic Broadcast message type as illustrated in [35, Fig. 4.5]

The *variables.go* file basically contains all the necessary global variables, which are initialized at the beginning of the execution. These variables are the number of servers and clients in the system, the maximum possible Byzantine processes, an indication of global or local execution, the ID of the process, the MVC, VC, SSVC default value, and the transient fault value for SSVC and SSABC.

4.4.3 Threshenc

In the *threshenc* folder, the signature and verification of messages take place. The *key_generator.go* file creates a secret key for each server which is used for further execution to sign and verify incoming messages. Even though the protocols do not need digital signatures to operate, the cryptographic process guarantees the validity of a server. That is, the messages between servers contain an appended digital signature of the message to make sure that the message is not malicious and therefore can be forwarded to the appropriate channels for algorithms' execution. This signature and verification process takes place in the *sign_and_verify.go* file as shown in Figure 4.7. The generation of the keys is completed before the execution of the atomic broadcast module; therefore, the only overhead of this procedure is the local file reading of the secret keys. The project design of using these digital signatures is followed as a virtue of the common system usage of the one developed in a previous thesis [35].

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

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

Figure 4.7 Sign and Verify methods as illustrated in [35, Fig. 4.6]

4.4.4 Messenger

The *messenger.go* is responsible for the communication of the system. Every necessary socket is initialized in that file, as well as the channels that receive the incoming messages. In addition, every function that transmits messages to other sockets or even broadcast a message to everyone, lies there. The binding of clients and server sockets happens at the initialization of messenger along with *goroutines* waiting to receive messages. Incoming messages are received as bytes, and they are transformed to the *message* struct to be handled appropriately. The newly received message is evaluated based on its type, it is transformed again to the correct message *struct* and it is forwarded to the corresponding protocol channel. The reverse procedure is also handled in messenger, as protocol messages are wrapped in a common message structure and then serialized to bytes to be sent with ZeroMQ. Finally, the Byzantine modifications of the values to be sent to other processes take place in a few functions in *messenger.go*. According to the scenario, Byzantine messages have either the same value, different values according to the recipient of the message, or are not sent at all in the idle scenario. The message modifications in the executions where Byzantine processes communicate with others affect every message sent in the whole protocol stack. That is, a certain value is sent to every process from binary consensus to atomic broadcast, both for the original and self-stabilizing executions. These Byzantine alterations take place right before broadcasting a message to others.

4.4.5 Modules

Every algorithm presented in Chapter 3 is implemented in the *modules* package and can be viewed on my GitHub repository [4]. The binary consensus algorithm uses a boolean value, which is based on whether its round number is odd or even, as the common coin to provide randomness. This is not Byzantine tolerant but it does not affect the effectiveness of BC as Byzantines do not attack the common coin. Reliable broadcast is included in this package and as mentioned earlier, it is utilized by every non-self-stabilizing algorithm to share its messages with others. Moreover, BC, MVC, and VC need to combine two numbers, their operation id and round of execution, to provide their service correctly. Cantor's pairing function [11], $(a^2 + 3a + 2ab + b + b^2) / 2$, is applied to merge the two numbers uniquely. The module that handles client requests lies in this package, too. Basically, when a request arrives, the *request_handler.go* receives the message from the messenger module and is responsible to call the atomic broadcast function for that request. Atomically delivered values are sent to *request_handler.go* and from that file they are forwarded to messenger to be provided as replies to clients.
Regarding the self-stabilizing implementation, both of the newly designed algorithms embed a reliable broadcast to their main execution body. To achieve that, the gathered messages are stored in a Go *map* type, called *msg*, where each process has its own *init*, *echo*, and *ready* messages. These messages are all the values that were already sent or the new to be sent in the next broadcast. Go's *map* type is a dictionary which matches a key-pair value. Furthermore, the message tuples stored in a process' message type, e.g. *msg[process_id][message_type]*, are saved as sets. That is, duplicates are not allowed, so before inserting something a check on its existence takes place. A value is inserted only if not found. This helps with further checks that must be made, such as whether only one message is inserted in the *init* section or if multiple messages with the same sender exist in the same

message type, i.e., a message conflict exists. These investigations are performed faster by having to deal with sets as the only thing to be examined is the number of messages in a particular message type. Namely, when two messages exist, which have the same sender, a violation has occurred. Both protocols execute at most one instance of the stack algorithm beneath them at any given time. The objective of this is to avoid perplexing incoming consensus decisions and to make the handling of their stored messages easier. While a lower-level protocol is executing in a separate *goroutine*, servers continue to broadcast their messages to avoid process crash if *goroutine* crashes. This happens also to avoid the situation of several servers not being able to execute deeper stack protocols, while others are stuck in a lower-level algorithm and wait for more servers to join, i.e., deadlock. This could take place if some servers have entered a beneath layer's execution, whereas others have cleared their *msg* values, as a virtue of a transient fault, after the last broadcast of the servers executing the lower consensus algorithm. In this scenario, some servers do not have the appropriate messages to reliably deliver any value in order to execute the lower-level consensus module, so they just wait for non-existent messages to arrive, which leads us to deadlock.

Self-stabilizing Atomic Broadcast has some additional implementation details. To be more precise, the SSABC *num* value is declared as unsigned integer to prevent transient faults altering the sign of the variable and to be a bounded counter for self-stabilization purposes. Each server handles at most one request at a time which implies an inspection on whether the server is free of requests. SSABC signals the request handler that a request can be inserted when it does not have one for delivery, otherwise the request handler waits for permission to add the new to be delivered message. In addition, the response of the algorithms is different if the transient fault value is returned from a lower stack level to them. Self-stabilizing Vector Consensus just returns the transient fault signal to the upper layer, whereas Self-stabilizing Atomic Broadcast continues with execution without delivering anything, thus a new SSVC instance will be called in the next round of the infinite loop. Finally, to atomically deliver the appropriate values in a deterministic order, an ascending sorting of the byte requests takes place. Therefore, the requests are delivered identically by every server.

4.4.6 Faults

For the self-stabilizing versions of vector consensus and atomic broadcast, the presence of functions altering the values of stored messages in *msg* is essential to simulate Byzantine behavior and transient faults. Therefore, the *faults* package contains a modification for each message tuples' set. The modification is similar for both protocols so only the changes of the

SSABC are shown in Figure 4.8 and 4.9. The whole code, however, can be found on GitHub [4].

Basically, for the Byzantine modification in Figure 4.8, a complete message set is created, which has a message for each process in the system. Every value contained is equal to the byte of “0” character. This happens for Byzantine processes to have the ideal message set and try to confuse correct processes with values suggested even for correct servers. Additional message changes according to the scenario take place in messenger.

Furthermore, the transient faults implemented alter any message in the whole *msg* set. The tuple values are changed to the byte of “0” character, while senders are changed to current IDs plus one. The atomic broadcast modification alters the *num* section of the messages to a specific number. These changes are done to a process according to a probability *p*, which is given as input. That is, if a server’s floating precision number, which is generated by a pseudo-random function in the range of [0, 1), is less than *p*, then its *msg* is modified. Moreover, functions assessing the clearance of the transient faults have been developed for debugging and testing purposes. These methods are used in validation unit tests to evaluate whether a process has cleared a transient fault, thus achieving a safe state.

```
func ByzantinevaluesSSABC(msg map[int]map[string][]ssabcMT)
map[int]map[string][]ssabcMT {
  // change values to their corresponding sending ones
        halfScenario := config.Scenario == "HALF_&_HALF"
        for i := 0; i < variables.N; i++ {
   msg[i]["init"] = []ssabcMT{}
        msg[i]["echo"] = []ssabcMT{}
        msg[i]["ready"] = []ssabcMT{}


                valueToSend := "0"
                if halfScenario {
                        valueToSend = strconv.Itoa(i % 2)
                }
                msg[i]["init"] = []ssabcMT{{Sender:i,Num:0,Value:[]byte(valueToSend)}}
                for _,t := range []string{"echo", "ready"} {
                        for j:=0; j<variables.N;j++{
                                msg[i][t] = append(msg[i][t],
ssabcMT{Sender:j,Num:0,Value:[]byte(valueToSend)})
```

```
                        }
                }
        }
  return msg
}
```

Figure 4.8 Self-stabilizing Atomic Broadcast message set's Byzantine modification [4]

```
func CreateSSABCTransientMsg(initFault, echoFault, readyFault, senderFault,
   valueFault, numFault bool, msg map[int]map[string][]ssabcMT, p float64)
map[int]map[string][]ssabcMT{

        s := rand.NewSource(time.Now().UnixNano())
        r := rand.New(s)

        if r.Float64() < p {
                faultTypes := []string{}

                // init
                if initFault {
                  faultTypes = append(faultTypes, "init")
                }
                // echo
                if echoFault {
                  faultTypes = append(faultTypes, "echo")
                }
                // ready
                if readyFault {
                  faultTypes = append(faultTypes, "ready")
                }

                // transient fault occurence
                for _, ftype := range faultTypes {
                        logger.OutLogger.Print("SSABC: ",ftype," transient fault:\n")
                        for id:=0;id<variables.N;id++{
```

```
                for i,_ := range msg[id][ftype]{
                    if senderFault{ // change sender
                        logger.OutLogger.Print(id," sender --> sender + 1\n")
                        msg[id][ftype][i].Sender = msg[id][ftype][i].Sender+1
                    }
                    if valueFault{ // change value
                        logger.OutLogger.Print(id," values --> all '0'\n")
                        msg[id][ftype][i].Value = []byte("0")
                    }
                    if numFault{ // change num
                        logger.OutLogger.Print(id,"num--> all MaxUint32/2\n")
                        msg[id][ftype][i].Num = math.MaxUint32/2
                    }
                }
            }
        }
    }
    return msg
}
```

Figure 4.9 Self-stabilizing Atomic Broadcast message set’s transient fault modification [4]

4.5 Client Implementation

The client implementation, as shown in Figure 4.10, contains almost the same components as the server’s one. Similar procedures, like the servers’ ones, are followed with the beginning of the execution of clients’ code. These include socket, global variables (in *variables* file), and *goroutines* for message exchanges initialization. For compatibility purposes, the client behavior utilized is identical to the one used in a previous thesis [35].

BFTWithoutSignatures_Client

- app
    - client.go
- messenger
    - messenger.go

- config
    - ip.go
    - local.go
- types
    - client_message.go
    - reply_message.go
- variables
    - variables.go
- logger
    - logger.go
- main.go

Figure 4.10 Client project structure as illustrated in [35, Fig. 4.7]

The folders *config*, *logger*, *types*, *variables* and *messenger* serve similar purpose as their server counterparts, despite the fewer number of functions in modules. The *variables* and *logger* packages are identical with the server ones and the *types* folder only contains the *struct* of a client message. The *config* module is similar to the one that servers use, with a small difference of missing the transient scenario set-up as clients are not prone to transient faults like servers. The *messenger* module developed is like the servers' one, but it misses the Byzantine modification functions. The methods needed for communication are implemented in *messenger* for message exchange capabilities between servers and clients, i.e., requests and replies.

The major difference of the client implementation in comparison with the server code is their behavior. In Figure 4.11, the behavior of client processes is presented. Clients' actions are defined in the *app* package in which clients handle requests and replies. Before sending any request, each client blocks for a few seconds to avoid the simultaneous request arrival to servers from every client. The blocking time is defined by client's ID number; thus it is unique for each one. Consequently, clients create a request, which contains a randomly selected character chosen by a pseudorandom Go function, to be atomically delivered and it is sent to *f+1* servers, to ensure a correct recipient of the request. Then, every client waits to get *f+1* ACK messages from different servers to be confident that the request was indeed successfully delivered, before sending the next one. In total, two different character values are sent from each client. The client keeps track of the response time of the *f+1* servers and runs indefinitely or until it is shut down.

```
var (
	runes = []rune("!\"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrs
tuvwxyz{|}~")

	replies  = make(map[int]map[int]bool) // num, from
	accepted = make(map[int]bool)		// if this num is accepted

	// Client metrics regarding the experiment evaluation
	sentTime  = make(map[int]time.Time)
	OpLatency = time.Duration(0)
	num = 0
	Total = 0
)

func Client() {
	rand.Seed(int64((variables.ID + 3) * 9000))		// Pseudo-Random Generator
	time.Sleep(time.Duration(variables.ID%10) * time.Second) // Wait a bit before
sending 1st request

	sendRune()

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

			// If more than F+1 with the same value, accept the array.
			if len(replies[message.value]) >= (variables.F+1) &&
!accepted[message.value] {
                                        accepted[message.value] = true
                                        OpLatency += time.Since(sentTime[message.value])
                                        Total++
                                        logger.OutLogger.Print("RECEIVED ACK for ",
message.value, " [", time.Since(sentTime[message.value]), "]\n")

                                        if num < 2 {
                                                  sendRune()
                                        }
                              }
                    }
          }()
}

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

          for i := 0; i < (variables.F + 1); i++ {
                    to := (randServer + i) % variables.N
                    flag := messenger.SendRequest(message, to)
                    if !flag {
                              randServer = rand.Intn(variables.N)
                              i--
                    }
          }

          sentTime[num] = time.Now()
}
```

Figure 4.11 Client behavior as illustrated in [35, Fig. 4.8]

Chapter 5

Experimental Assessment

5.1 Validation Unit Tests

For the correctness assessment of the studied algorithms in specific scenarios, several unit tests regarding various situations have taken place. The protocols must be evaluated in the presence of Byzantine faults to estimate the influence of Byzantines nodes in the consensus, and transient faults for the convergence property. Besides this, scenarios without transient faults reveal the satisfaction of the self-stabilizing closure property as the algorithm should not deviate from its safe state. Thus, it has to preserve the properties of the consensus problem. The following Sections present all the unit tests that were used.

Furthermore, the correctness of the algorithm was assessed by comparing the decision value of each process. Satisfying the consensus or atomic broadcast properties is the desired way to indicate the error-free execution of the protocols. If every non-faulty server holds the same decision at the end of the experiment, then the algorithm was proven correct in that specific unit test. This check has taken place both in transient prone and in transient free environments, regardless of the existence or behavior of Byzantine processes.

The evaluation of the convergence property, however, was done differently. We know the altered values of a message as a virtue of the manually created transient faults' contents mentioned in the following Sections. Therefore, the check of whether the algorithm was able to clear the transient faults, i.e., return to a safe state, was done by evaluating the content of the message tuples. When none of the messages stored in the *msg* array contained a value which was created by a transient fault, it means that the process has managed to clear these temporary errors. Namely, the amount of messages that is affected by a transient fault is the number of messages in *msg* that match the transient fault values used in the experiment. Consequently, the clearance of transient faults in every process manifested the capability of the algorithm to converge in that specific unit test.

In addition, several validation tests have taken place both for self-stabilizing atomic broadcast and vector consensus algorithms. Their correctness and convergence properties were evaluated as defined above. More specifically, every unit test mentioned below has assessed the behavior of the self-stabilizing atomic broadcast, whereas each one except from the *num* based transient faults has evaluated the actions of self-stabilizing vector consensus. Both protocols were successful in completing their corresponding unit tests in terms of correctness and convergence.

5.1.1 Failure-free Unit Test

The base unit test consists of a system with neither Byzantine processes nor transient faults. It is mainly focused to evaluate whether the processes behave correctly with the proposed algorithm. This is considered as the ideal working scenario and can serve us by having the metrics of this scenario as a base for comparisons with other more complicated ones. Another reason to have this unit test is to compare the results and measurements of the studied algorithms with other algorithms proposed in the bibliography, especially with the non-self-stabilizing version of the protocol.

5.1.2 Byzantine-only Unit Tests

The proposed algorithm ought to be tested with Byzantine processes, acting in various ways, to evaluate the behavior of the system when processes that act arbitrarily exist. The scenarios presented below were developed to assess these kinds of faults. An important thing to mention is that the changes of the values sent by Byzantine processes are implemented as modifications of the messages just before broadcast. For the sake of simplicity, but not affecting the effect of the scenario, the values sent from Byzantines are 1-byte characters, either “0” or “1”, for every message stored in their array. For example, if a Byzantine process was supposed to send an *echo* and a *ready* message with another process as the sender in the message tuple, the value of every message is overwritten with the 1 byte one.

5.1.2.1 Byzantine Idle Scenario

This is another crucial test to gauge the behavior of the system with crash faults. In this case, Byzantine processes do not send any messages throughout the execution of the algorithm, so the only processes that send messages are the *n-f* correct ones. The absence of the Byzantine processes in the consensus procedure can show the tolerance of the algorithm when having *f* idle participants.

5.1.2.2 Half and Half Attack Scenario

Half and Half scenario considers the manipulation of the messages by Byzantine processes and their effort to confuse correct processes by sending different values to them. More specifically, Byzantine processes send a specific value to half of the correct processes and a different one to the other half. They are coordinated to send the same value to each process, so correct nodes receive a specific value from every faulty process. In detail, any message send from BC to SSABC is modified based on the ID of the receiver – the “0” character is sent as one byte to servers with even IDs and the “1” character is sent to those with odd IDs.

5.1.2.3 All Attack Scenario

This scenario is an expansion of the previous Half and Half scenario and has a greater impact on the confusion of correct nodes. Byzantines send coordinately a specific byte, the “0” character, throughout the consensus stack for every outgoing message. This creates greater support for Byzantine proposed values and achieves the maximum confusion of correct nodes. The BFT nature of the algorithms, however, ensures that correct processes cannot be affected by the coordinated attack of the Byzantine nodes.

5.1.3 Transient Faults-only Unit Tests

The algorithms studied in this thesis are self-stabilizing, so scenarios having transient faults must be considered for the appropriate evaluation of them. The following scenarios ought to test that with faults disturbing majorly the memory of a process, namely changes in every saved message, the protocol continues to execute as intended.
When a transient fault takes place, the properties of the consensus problem cannot be ensured because the system is not in a safe state. Nevertheless, the algorithm must converge to a safe state, in finite steps, and continue its safe state execution without ever deviating from it. With a focus on convergence, the following unit tests were executed and tested the ability of the system to expel the transient faults. To achieve that, a module testing the saved messages of a process was implemented, which checks for each message if the simulated transient fault is present. That is, it evaluates whether every message carries the correct value, based on the sender in the corresponding testing scenario, and whether the sender of the message has an ID which belongs to the ID range of the system. In SSABC scenarios, stored tuples are tested also for the correct values of their *num* section. This module is not part of the algorithm, and it was just used for debugging purposes to evaluate the convergence property.
Transient faults can change any part of a message tuple saved by a process. Every scenario mentioned below was executed with the transient fault affecting every process of the system or just half of them, based on a given probability. The faults used, however, affect every message in each category. After transient clearance, the test continued its execution to

evaluate the protocols' correctness in transient prone systems, i.e., if processes decide the same values.

5.1.3.1 Value Transient Fault Unit Tests

Value transient faults can affect the convergence of the algorithm as they can lead to wrong content of the *msg* array. The remainders of these faults in the system means that the studied algorithm is not transient fault resilient. The studied algorithm must be able to eventually clean up the system from these faults and to be able to continue the safe execution with values that are proposed by processes of the system. For simplicity purposes, but without affecting the impact of the scenario, value contaminant transient faults overwrite the value of every affected message with the "0" 1-byte character.
The above-mentioned test was held multiple times to affect all *init*, *echo*, and *ready* messages separately and messages of any other type' combination possible.

5.1.3.2 Sender Transient Fault Unit Tests

The goal of this scenario is to evaluate the capability of the algorithm to eliminate messages with senders' IDs that do not correspond to the consensus participant processes. Transient faults can alter the ID of a sender in a message tuple, which can lead to several problems of the expected execution. These messages should be removed from the system to achieve a safe state. In detail, two-unit tests were executed with sender altering transient faults. In the first, the sender part of the message tuple was changed from sender ID to sender ID+1 and in the second sender ID-1 was the transient sender value. This is sufficient to test the behavior of the algorithm in these situations.
The above-mentioned test was held multiple times to affect all *init*, *echo*, and *ready* messages separately and messages of any other type' combination possible.

5.1.3.3 Num Transient Fault Unit Tests

This experiment targets the self-stabilizing atomic broadcast protocol as it is the only algorithm where its message tuple contains a *num* section. Basically, the *num* of each message is changed and set to a specific value. The objective of this is the evaluation of convergence in this kind of transient faults. The studied protocol is able to circumvent the disturbance while moving on to a safe state execution. The above-mentioned test was held

multiple times to affect all *init*, *echo*, and *ready* messages separately and messages of any other type' combination possible.
Moreover, separate experiments were done to assess the behavior of SSABC when the value of its local counter is changed. These changes regard the altering of the counter to a lower or greater value compared to the correct number, at that specific time. In any situation, the protocol was able to circumvent the alterations and managed to reach a safe state.

5.1.3.4 Full Tuple Transient Fault Unit Tests

This unit test, which contains the most distractive and general kind of transient fault, provides the most difficult assessment of the algorithms as every fault mentioned in Section 5.1.3 occurs simultaneously. These faults alter the values of every stored message based on the types of messages that are affected by the transient failure. In detail, the sender part of a message is changed to sender ID+1, the value part is altered to a constant byte, namely the "0" character, and if the message tuple contains a *num* component it is set to a constant value. Fundamentally, this transient fault completely changes the state of the system with messages that have the required support for their existence. For instance, the existence of *ready* messages is supported by *init* and *echo* messages as all of them contain the same value. Despite the severity of this scenario, the algorithms can restore system's safety by vanishing faulty messages.
The above-mentioned test was held multiple times to affect all *init*, *echo*, and *ready* messages separately and messages of any other type' combination possible.

5.1.4 Combined Byzantine and Transient Faults Unit Tests

The system developed must be able to withstand both transient faults and Byzantine processes simultaneously. In this scenario, Byzantine processes act arbitrarily while the transient unit test of Section 5.1.3.4 takes place with effect on every message stored. These unit tests basically assess every aforementioned Byzantine behavior with the Full-Tuple transient fault. To make things tougher, transient faults alter the values of messages to the same values sent by Byzantine processes. Despite the severity of these scenarios, the algorithms are able to tolerate these faults by reestablishing the system's safety and reaching a consensus while satisfying the properties mentioned in Section 2.3.

5.2 Experimental Environment

The experiments that assessed the behavior of the algorithms were concluded in a real-world environment. More specifically, a cluster of five machines was used to simulate processes' actions in an asynchronous distributed system. All machines run the protocols in parallel, while communicating with each other by using their public IP and ZeroMQ library. Machines' specifications are presented in Table 5.1. In detail, there are two machines with quad-core CPUs, while others have single-core or dual-core CPUs. According to the scenario, every machine run one server instance, except from the two quad-core ones, 0 and 1, which run the extra server instances when the number of servers exceeded four. A different machine handled all client requests, but when the amount of clients was 50, an extra machine was allocated for client behavior, with each machine having 25 client processes. That is, the number of machines used for requests in relation to the number of clients in the system is the following:

- Client number ≤ 25 → one machine.
- Client number equal to 50 → two machines.

In the experiments with multiple client machines, servers' instances were executed on our two best machines. Specifically, in these assessments, only four servers were used therefore each one of the quad-core processors had two active server instances.

Machine Number	**CPU(s)**	**Threads per core**	**Core(s) per socket**	**Socket(s)**	**CPU Model**
0	8	1	4	2	Intel™ Xeon™ CPU E5320
1	8	1	4	2	Intel™ Xeon™ CPU E5320
2	1	1	1	1	AMD Opteron™ Processor 252
3	2	1	1	2	AMD Opteron™ Processor 246
4	1	1	1	1	AMD Opteron™ Processor 252

Table 5.1 Cluster machines' specifications

5.3 Assessment Metrics

The studied algorithms must be evaluated on various aspects to conclude their general behavior in experimental scenarios. These measurements should assess every theoretical complexity component for distributed protocols. Therefore, for every experiment taken the evaluation metrics consist of three aspects:

- **average operation time**: time needed by servers to atomically deliver a client request. It is measured by clients as the time between the transmission of their request to the servers until they receive *f+1* ACK replies.
- **average message complexity**: the amount of messages exchanged between servers to correctly atomically deliver a set of values in one atomic broadcast round. Measured as the total number of messages sent to cover every request divided by the number of SSABC rounds, i.e., the amount of consensus calls made in which requests were delivered.
- **average message size**: the size of a message sent in one atomic broadcast round in MBs. Measured as the cumulative size of the exchanged messages divided by the total number of messages sent.

Scenarios with transient faults have been tested for **convergence time**, too. That is defined as the time between the last transient fault and the atomic delivery of a correct message. In our unit tests for convergence quantification, the system starts from an arbitrary state, therefore convergence time is the time the system needs, from its initialization, to deliver its first correct request.

None of the measurements taken regard Byzantine processes in their calculation as we want to evaluate only the behavior of the correct processes of the system and not arbitrary behaving ones. Moreover, in our experiments, clients' requests are already available to servers when they start their execution. Specifically, all the numerical assessments are taken from the moment clients' requests are available until the atomic delivery of all of them. This makes our measurements more robust as we do not consider initialization and starting network overheads.

5.4 Experimental Results and Analysis

The following experiments took into consideration different number of clients, servers along with Byzantine processes' actions, in each environment. The amount of clients and servers used in each scenario is indicated on each graph. Byzantines' behavior is also mentioned in the following figures. As a summary of the four different types of behaviors mentioned in Section 5.1, we have the following:

- **NORMAL**: no presence of Byzantine processes.
- **IDLE**: only crash failures exist in the system.
- **HALF and HALF (H&H)**: Byzantines send one value to even numbered processes and a different one to the oddly numbered servers.
- **ALL**: Byzantines send one value to any other process.

Moreover, two versions of the tests took place: one with transient fault presence and one without. When transient failures existed, every process was affected by a given probability *p*. That is, every process picks a random number in the [0,1) range and if it is less than *p*, then a transient fault affects the process. In the clients and servers scalability tests every process was affected (*p*=1). These faults altered the content of the messages in the *msg* array in the way described in Section 5.1.3.4. Every stored message was affected, and each tuple component was altered. The value and num parts of the message were set to a predefined constant value, whereas the sender component was changed to sender ID+1. They struck when the *msg* array contained at least one *ready* message for the first time. This provides a random transient fault occurrence to make its impact larger. Basically, the transient fault took place only once as the evaluation of a system prone to these failures considers the last transient fault and onwards.

In the following graphs, several abbreviations are mentioned. Here are their explanations:

- **T**: Transient fault probability, *p*.
- **S**: Servers used.
- **C**: Clients used.

The chance of a server to begin in an arbitrary state in convergence tests is defined as transient appearance probability. That is, if the probability is 1, every server is affected, but if it is 0.5 only half of them start with a faulty state.

5.4.1 Clients Scalability

These measurements take into consideration all the previously mentioned experiments. That is, a version of the graph with both Byzantine and transient faults exists and another one only with Byzantine faults. These tests have been executed with the same amount of servers, four to be exact. This was done to assess only the effect of soaring clients in the aspects mentioned in Section 5.3. Specifically, executions were completed with 1, 10, 25, and 50 clients.

5.4.1.1 Operation Time

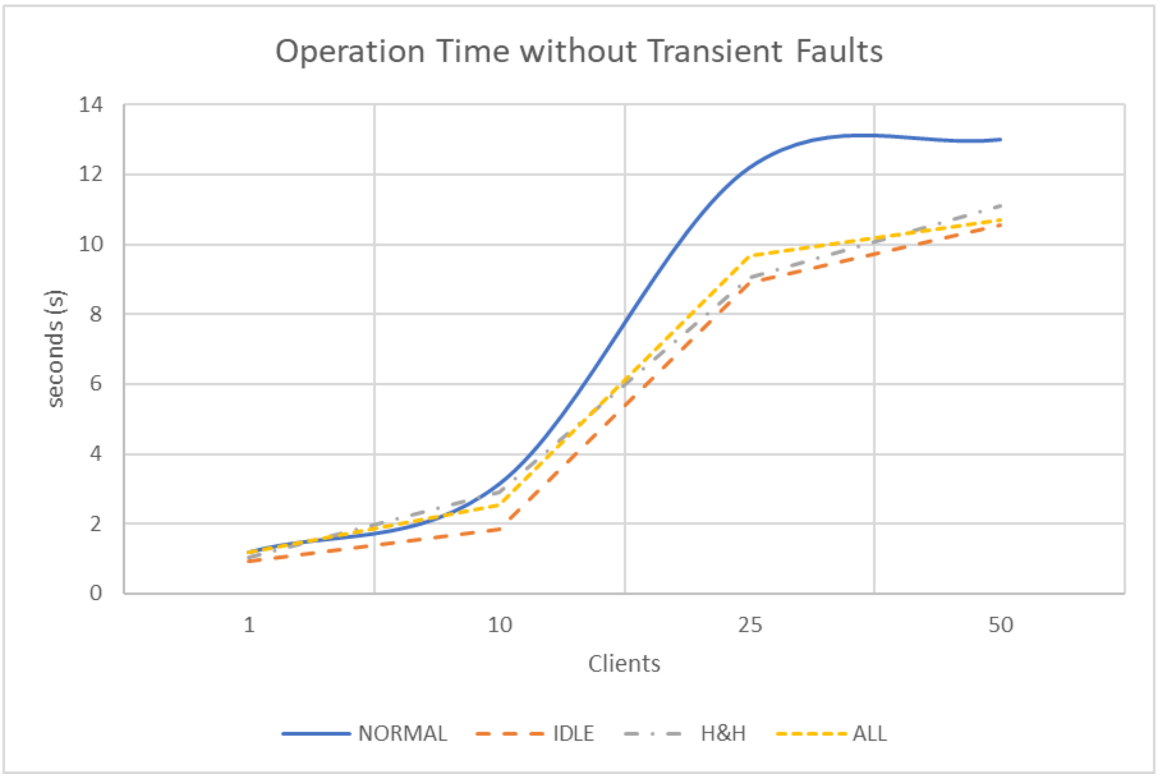


Figure 5.1 Operation Time - constant servers and increasing clients without transient faults

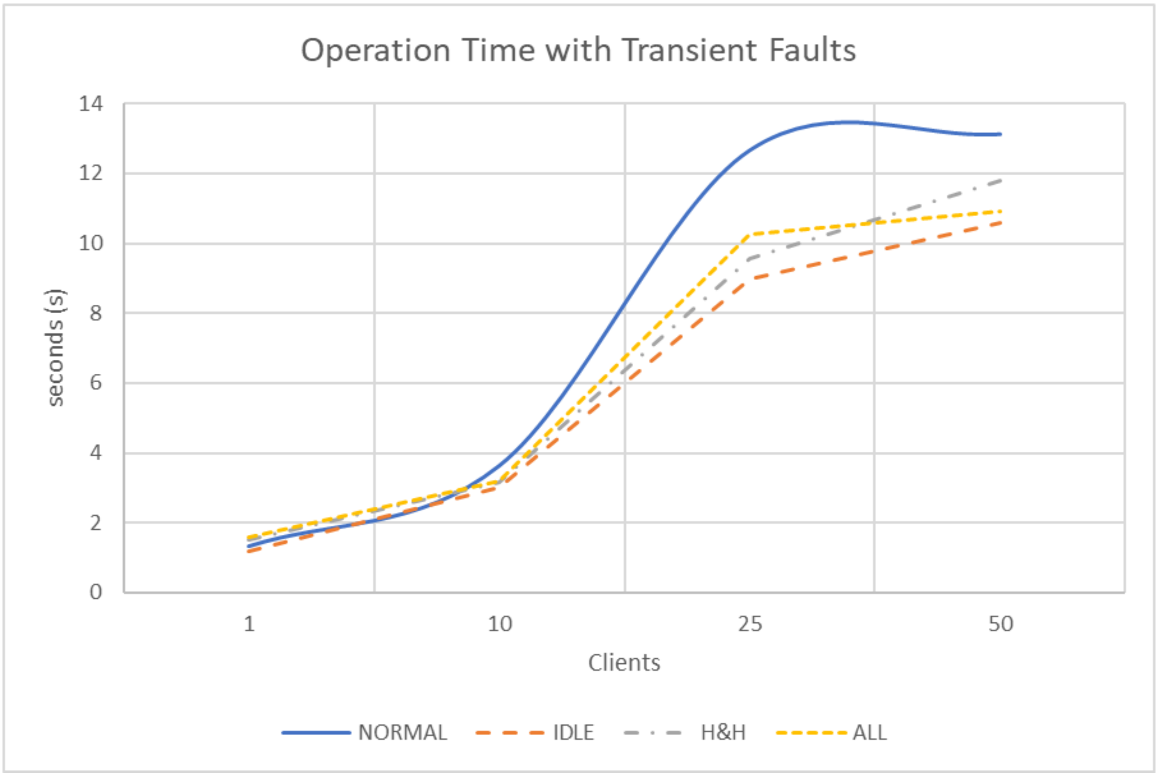


Figure 5.2 Operation Time - constant servers and increasing clients with transient faults

The operation time of the applied unit tests is shown in Figures 5.1 and 5.2. These regard the absence and presence of transient faults, respectively.
To begin with, in both graphs, the time needed to deliver every incoming request surges as the number of clients climbs. This is due to servers' proposal of at most one request, i.e., they cannot suggest a new request until they atomically deliver their current one. Therefore, as clients soar the same thing happens to the incoming requests. Due to a server's proposal of only one of them at a time, servers need more time to accomplish their task. The time escalates the most when we go from 10-clients to 25-clients simulations, as it gets tripled. This indicates a critical point of the system in which servers' work is at its maximum at any given moment.
The influence of Byzantine behavior does not affect dramatically the time in any experiment, but the time needed in the absence of Byzantines when the clients are larger than 10 is significantly more compared to scenarios with Byzantine failures. This is not a foreseeable result as the malicious actions of Byzantines target the confusion of correct nodes. However, this could be explained as the size of the values sent by Byzantines is notably less compared to the values of correct processes. This influence both the computation time of a server to evaluate the content of a message and the time required for a message to be sent over the network. To elaborate more, the influence of messages' size is more prone to appear on occasions where a lot more messages are sent to fulfill the requirements of the experiment. The reason behind this is that as more and more messages are smaller, the difference in time needed to handle only normal messages and several smaller ones cumulatively stacks up and eventually appears largely.
Additionally, in crash faults environments, there are less messages sent over the network and less values stored in *msg* to be checked. Therefore, the least operation time among any experiment with the same clients is when failed processes only halt indefinitely. The time needed in normal and transient prone environments is nearly identical for any experimental scenario. The slight difference between them exists as a virtue of the time required to flush and recompute values in several *msg* entries to circumvent the transient fault. It is also shown in practice that despite the influence of a transient failure, the protocol continues and completes its task in times that are almost inseparable to situations where no transient faults take place. This means that the studied algorithms effectively handle these corruptions and therefore could be applied in real-world scenarios. Furthermore, the slopes of the graphs are interchangeable, which further supports the small influence of tolerating a transient fault.

5.4.1.2 Message Complexity

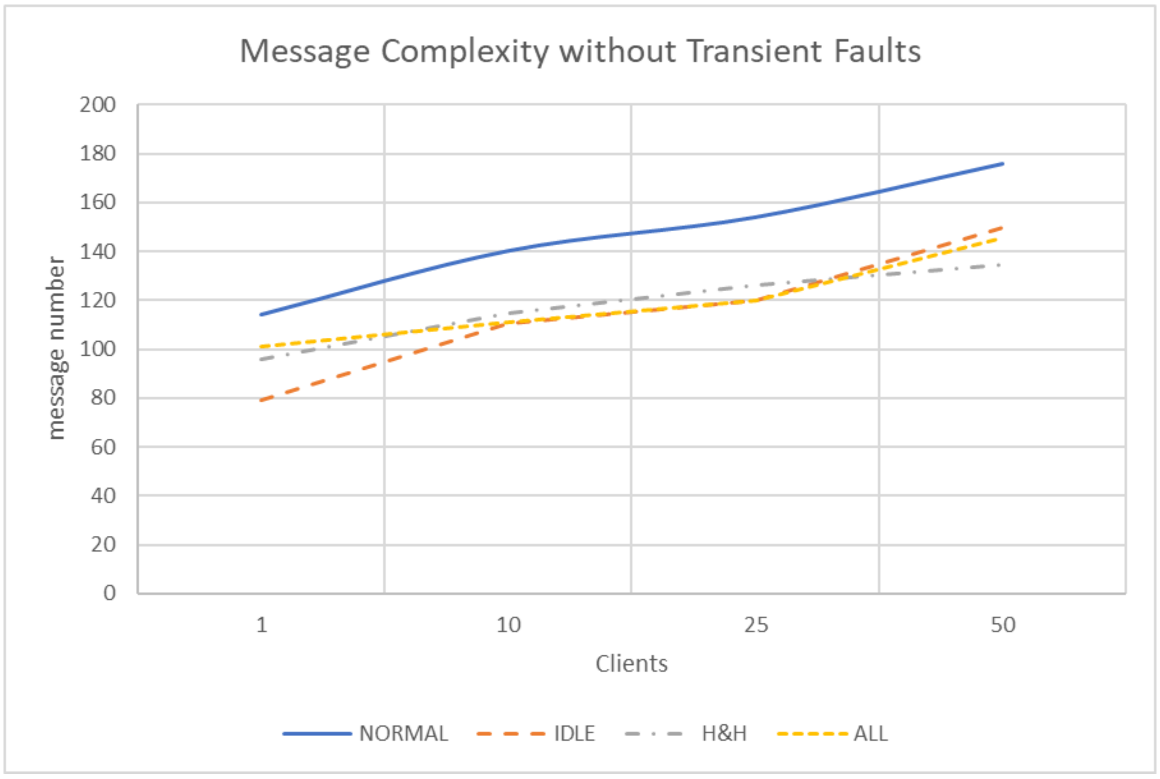


Figure 5.3 Message Complexity - constant servers and increasing clients, no transient faults

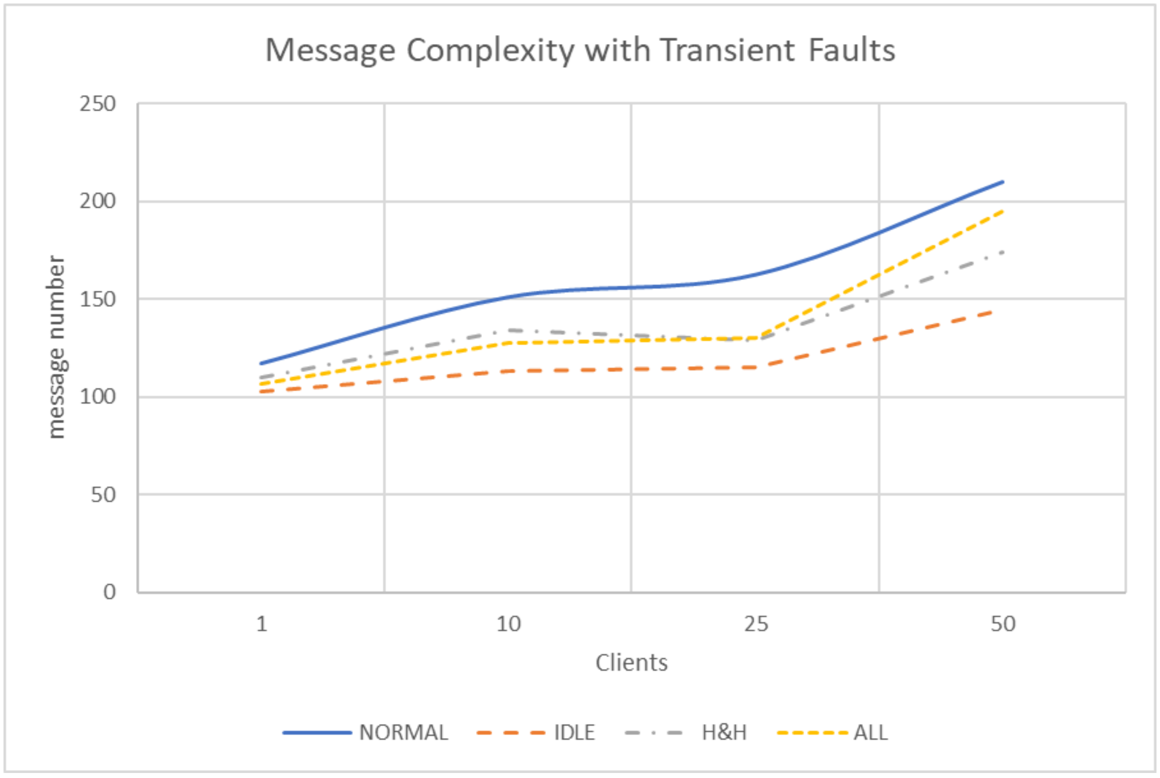


Figure 5.4 Message Complexity - constant servers and increasing clients with transient faults

Moving on to message complexity graphs, we observe different behaviors between the transient and normal scenarios. For clarification, in the self-stabilizing protocols processes broadcast messages in the *do-forever* body before proposing a value to the lower layer and while waiting to get a response from it. This means that more executions of lower stack protocols or more iterations of the *do-forever* loop result in more messages sent. This pattern is shown in both experiments, as indicated in Figures 5.3 and 5.4.

In both unit tests, the number of messages sent is increasing almost linearly as clients in the system soar. As mentioned before, the surge in incoming requests requires more iterations and executions of every algorithm in the stack which result the rise of messages exchanged. With more clients we have more incoming requests, thus the graph form is as anticipated. Moreover, the amount of messages sent is similar in transient and normal environments with slightly more values exchanged in systems with transient presence. This is completely normal as the extra messages are needed for clearing the incorrect values and for bringing the protocol back in its safe state track.

Additionally, in both environments the *NORMAL* scenario is the test with the most messages sent over the network. However, in transient free systems, a substantially larger number of values are exchanged. On the other hand, the least transmitted messages are measured in experiments where faulty nodes halt indefinitely. This is the case in both systems as *IDLE* tests are the lowest in almost any scenario. Furthermore, the metrics gained when Byzantine processes propose malicious values are almost identical when considering each environment separately, which is defined by transient presence or absence. This indicates a general stability of the protocol regardless of malicious actions.

5.4.1.3 Message Size

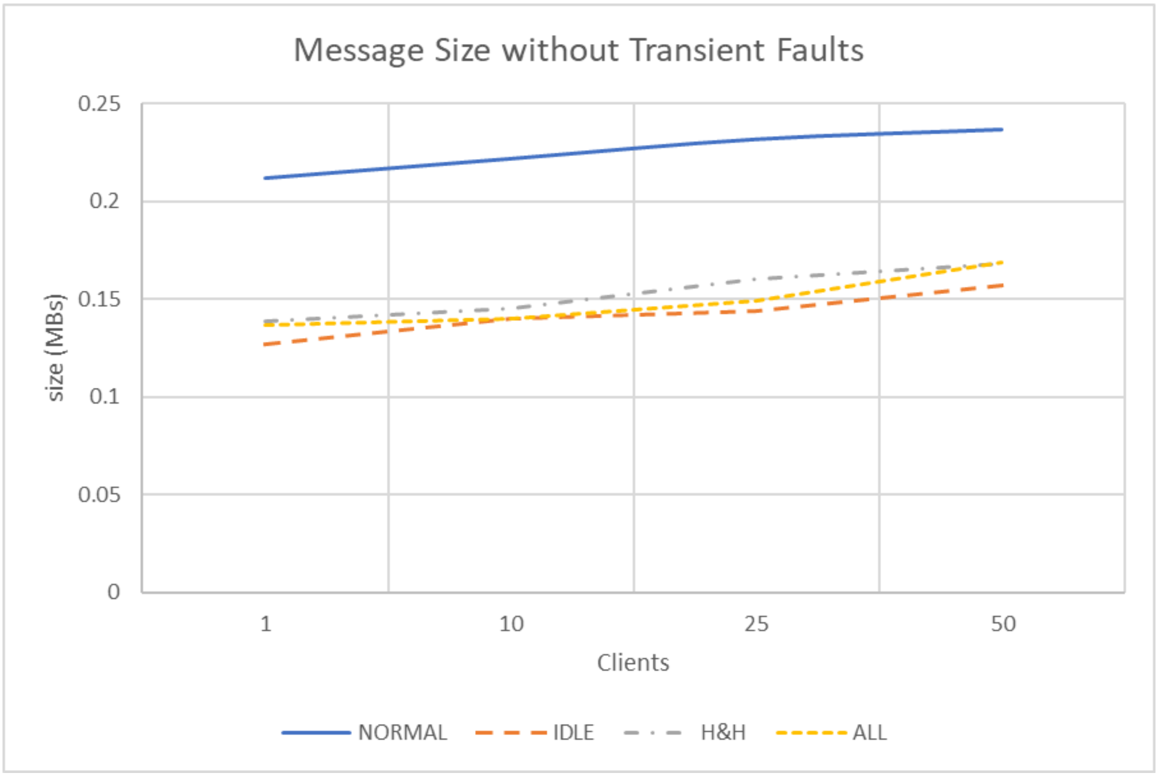


Figure 5.5 Message Size - constant servers and increasing clients without transient faults

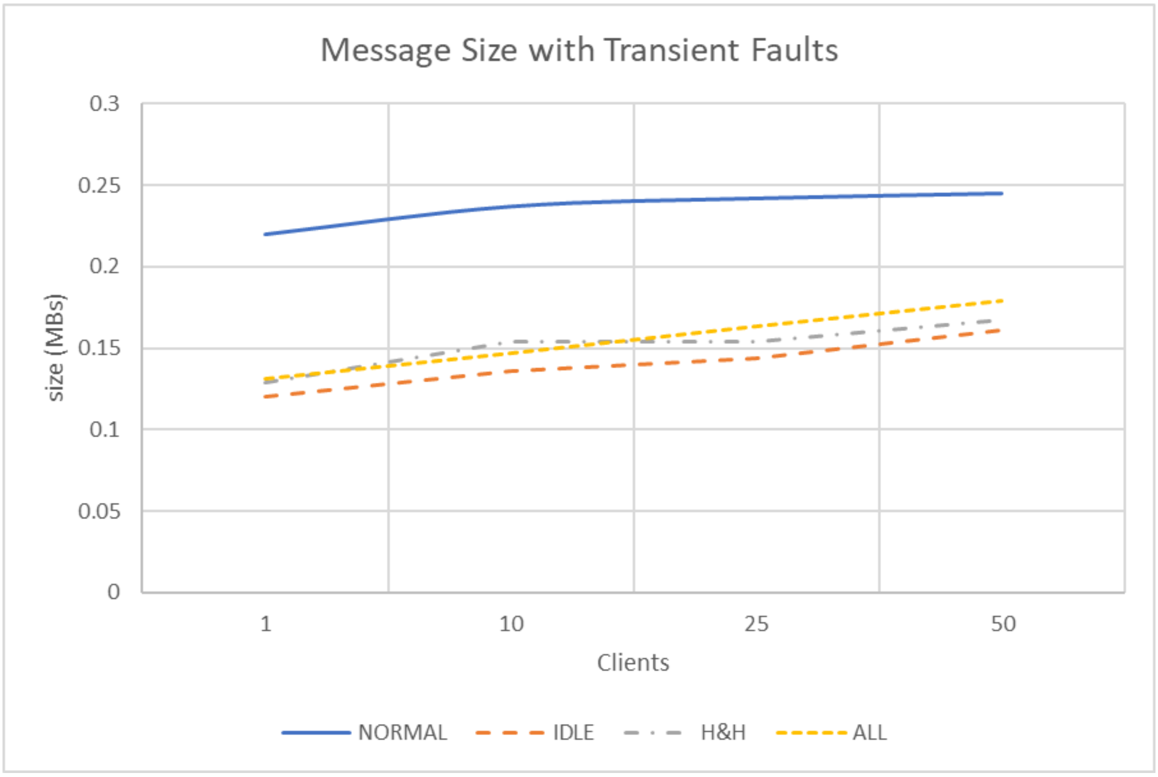


Figure 5.6 Message Size - constant servers and increasing clients with transient faults

Finally, average message size is shown in Figures 5.5 and 5.6. The message size is defined as the size of the *msg[i]* entry of p_i's array, which is basically what is broadcast to others. Since servers only propose at most one value, the messages in a process *msg* array are only affected by the amount of servers suggesting a value for delivery. Therefore, each *msg's* entry is bounded by the number of servers in the system. When fewer servers have a request to deliver, we expect smaller messages to be sent.

As expected, the size of the messages is almost constant as they fluctuate slightly in every line graph. This is due to the unification of the protocol messages where on every broadcast all the *init*, *echo*, and *ready* values of a process are sent over the network. Since *msg* entries are bounded and the number of the servers is persistent, the results are foreseeable. The fluctuation is seen in simulations with fewer clients. More specifically, in the one client test the message size is significantly less compared to the other scenarios. This is due to the fact that only some of the servers have a proposal as a virtue of the client's behavior to send requests to *f+1* servers, thus leaving some servers without one. This implies fewer messages in *msg* to be broadcast and therefore less average message size.

When more clients are utilized, servers constantly get a value to deliver from some client. This implies that in most of the atomic broadcast rounds all servers have a request to propose so the average size moves towards the value of a full *msg* entry, with either Byzantine values or only correct ones. In contrast, with fewer clients the exact opposite happens, and some broadcasts contain less than four values in messages, which lower the average. These two observations explain the slight increase of message size in environments with more clients.

Furthermore, as anticipated, the smallest and largest sizes occur in *IDLE* and *NORMAL* scenarios respectively. In unit tests with crash prone processes, there are less messages stored in *msg* entries as these nodes do not send any suggestions at all. Additionally, in invulnerable systems only correct messages are sent, which are larger compared to the ones sent by Byzantines. In detail, because of the altering of the Byzantine messages, which propose a one-byte value instead of a several byte correct message, several *msg* entries contain these smaller messages from faulty processes. The result of this is the shrinkage of the message to be broadcast as messages from Byzantine processes, which are forwarded by correct nodes, are smaller in size. Thus, in *NORMAL* experiments the average message size is significantly larger, while *HALF and HALF* and *ALL* scenarios' sizes are very similar.

Finally, as presented in the two systems, transient failures do not influence the measurements crucially. There are certainly more fluctuations in transient tests which are expected because of the nature of transient faults in our system. These faults alter the values stored in correct processes to a one-byte value. Thus, the average size is affected when a memory corruption takes place as it lowers the outgoing message size for a portion of the execution. Due to transient faults, however, more messages are needed to be sent for correct servers to eventually clear the corruption. Consequently, more messages are sent with requests from every server, either a new one or a delayed previous one. Therefore, if the execution time is not vastly larger compared to the convergence time, i.e., 1 client exists, the average size is lower as a virtue of smaller sized messages. Otherwise, the total average size is slightly larger because of the extra full *msg* entries sent for a longer period of time.

5.4.2 Servers Scalability

The following assessments take into consideration all the scenarios of Section 5.4. Namely, graphs for both only Byzantine failures and Byzantine with transient faults exist. These experiments have been taken with the same number of clients, which were 25. This was done to evaluate only the effect of soaring servers in the aspects mentioned previously. Specifically, executions were completed with 4, 7, 10, and 13 servers.

5.4.2.1 Operation Time

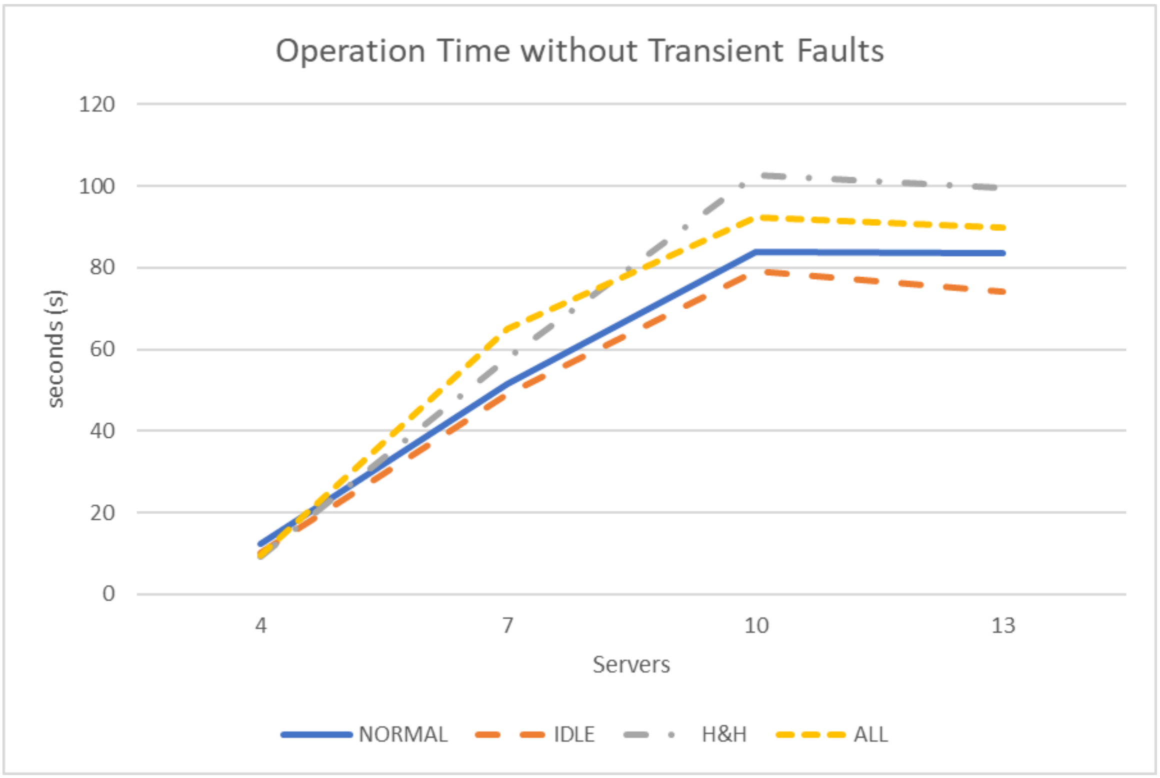


Figure 5.7 Operation Time – constant clients and increasing servers without transient faults

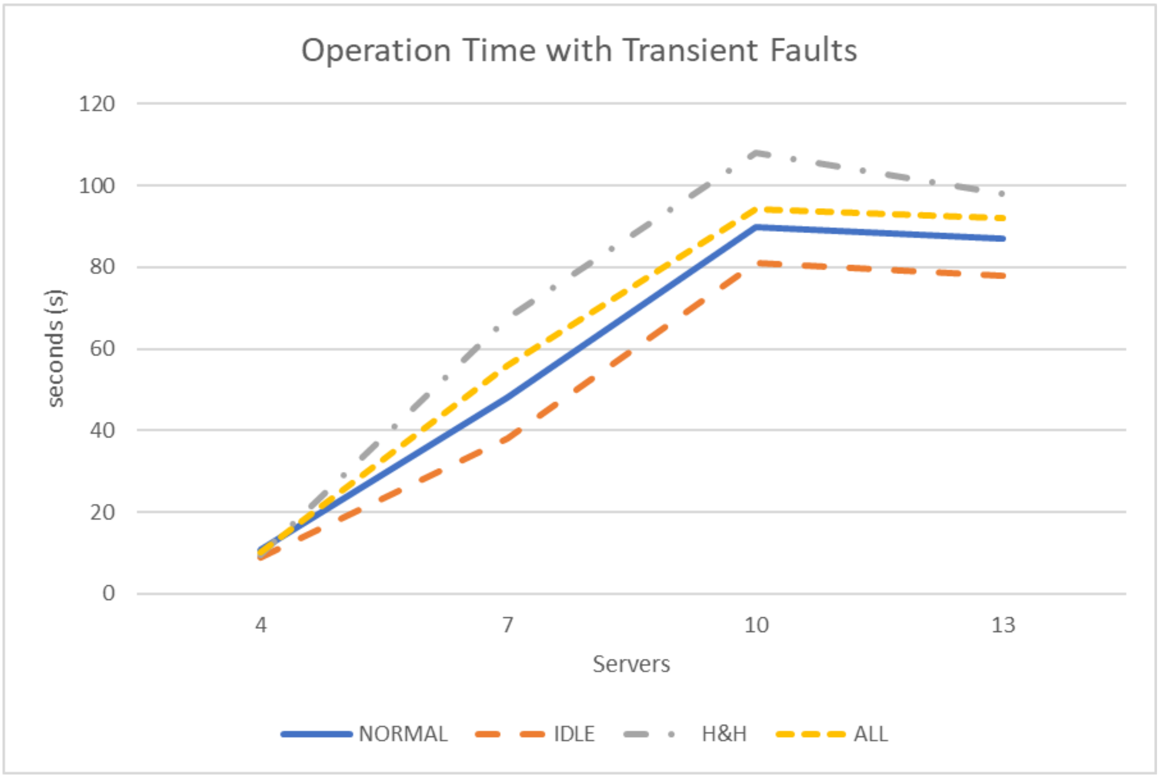


Figure 5.8 Operation Time – constant clients and increasing servers with transient faults

At first, the operation times of our observations are shown in Figures 5.7 and 5.8. There are many similarities between the environments with the absence and existence of transient faults.

Both unit tests indicate a positive association between server number and operation latency. This is anticipated as the time complexity of a *do-forever* iteration is $O(n^3)$, where n is the number of servers in the system. This implies that the time to deliver every request is larger as server number rises. One would expect that operation time would decrease as more servers exist in the system. With more servers, more requests can be proposed simultaneously for consensus. However, the time needed for the checks of the *msg* array circumvents the impact of more requests being proposed. This linear time climb occurs in both experiments up to a certain point, with thirteen servers, where the larger proposal set has more influence in time compared to smaller systems, therefore, disturbing the overall time increase.

Furthermore, the lowest time is recorded with crash failures, whereas the maximum time happens when Byzantines propose a malicious value for everyone. In crash prone environments, less messages, specifically n-f, are stored in each process' *msg* array so less time is required for the inspections to take place. In addition, the malicious values suggested by Byzantines confuse correct processes by making them flush incoherent values and refilling the gaps with new messages, which enlarge protocol time. These observations explain the time measurements. The presence of transient faults lowers moderately the operation latency in ten server or smaller systems as smaller messages, which exist due to transient faults, need less transmission time. In larger environments, transient faults increase slightly the operation time as a virtue of the extra flushes and messages sent to clear these temporary failures. In total, transient faults quantifications do not have significant distinctions compared to the transient free measurements.

5.4.2.2 Message Complexity

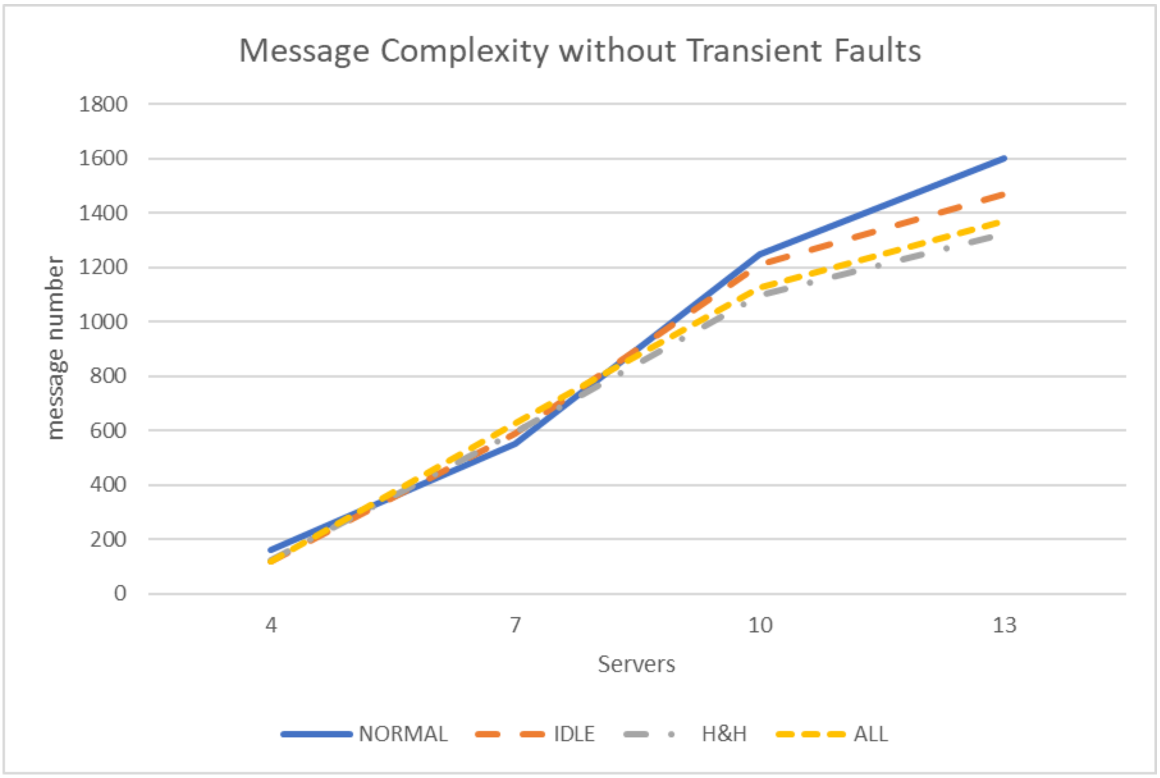


Figure 5.9 Message Complexity – constant clients and increasing servers without transient faults

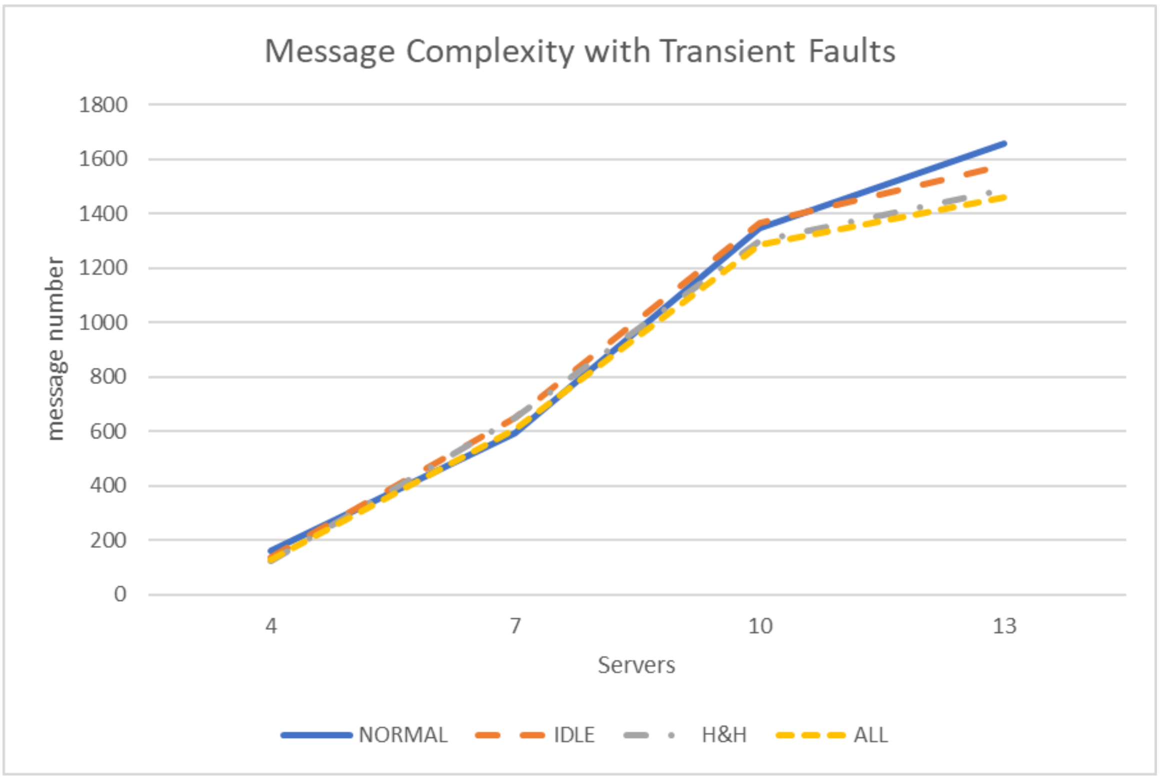


Figure 5.10 Message Complexity – constant clients and increasing servers with transient faults

Moving on to message complexity, both graphs indicate the expected experimental measurements. The amount of messages sent is affected by the number of servers in the system, and the iterations of the *do-forever* loop for broadcasts either in the main body of the loop or during the waiting for the consensus results.

At first, in either transient or normal environment, messages sent surge as servers' number rises. This is expected due to the message complexity's dependency on servers as the broadcast operation sends values to more recipients. However, this rising trend changes in the system with thirteen servers. Thirteen servers need less iterations of the *do-forever* loop to deliver all the requests, which are produced from a constant number of clients, thus, the broadcast module is called significantly less times. The fewer iterations needed are supported by the decrease of operation latency's trend in Figures 5.7 and 5.8. Consequently, the message complexity's trend gets lowered in such environments despite the increase of broadcasts' recipients.

Furthermore, the message measurements between any Byzantine behavior in each system separately are nearly identical. This presents the anticipated result that correct servers' actions are not defined by the behavior of malicious ones, therefore they exchange approximately the same amount of messages in any scenario. Some minor differences appear in crash prone systems as a virtue of more messages sent due to the slightly less time needed for checking the *msg* array. Moreover, in transient environments marginally more messages are required as some extra values are exchanged to bring the system back to a safe state. This happens to overwrite incorrect messages, which were transmitted earlier.

5.4.2.3 Message Size

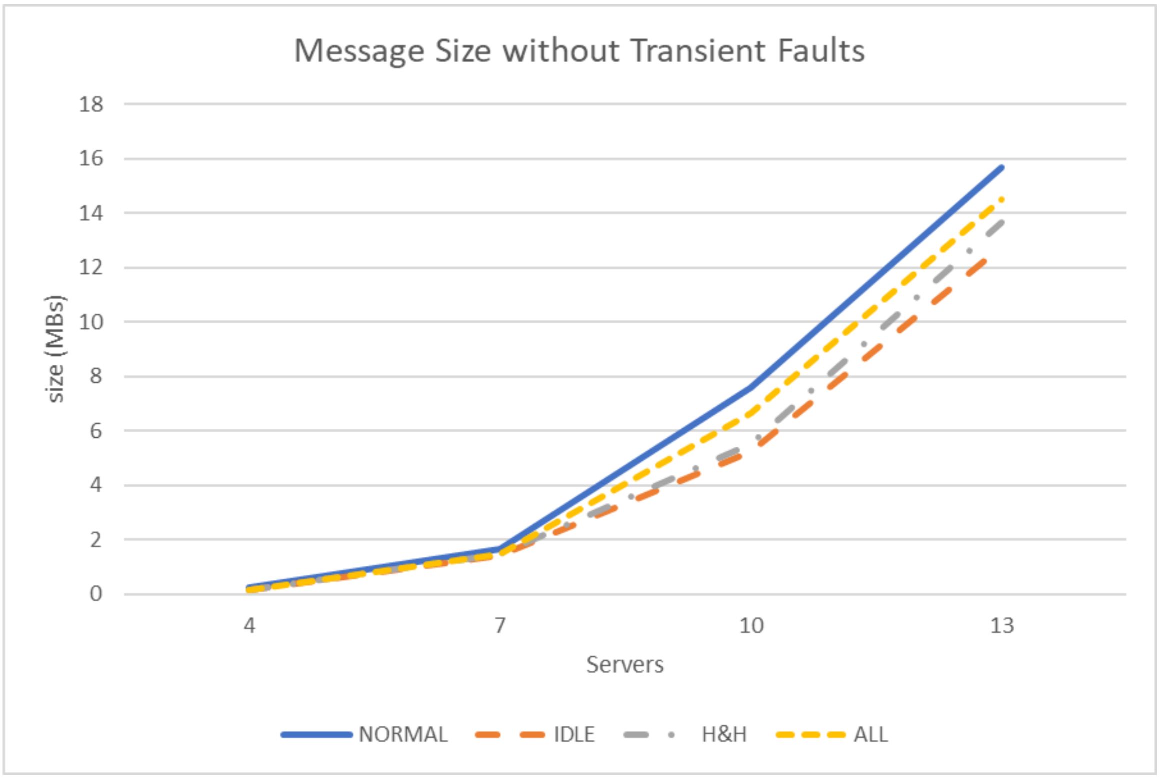


Figure 5.11 Message Size – constant clients and increasing servers without transient faults

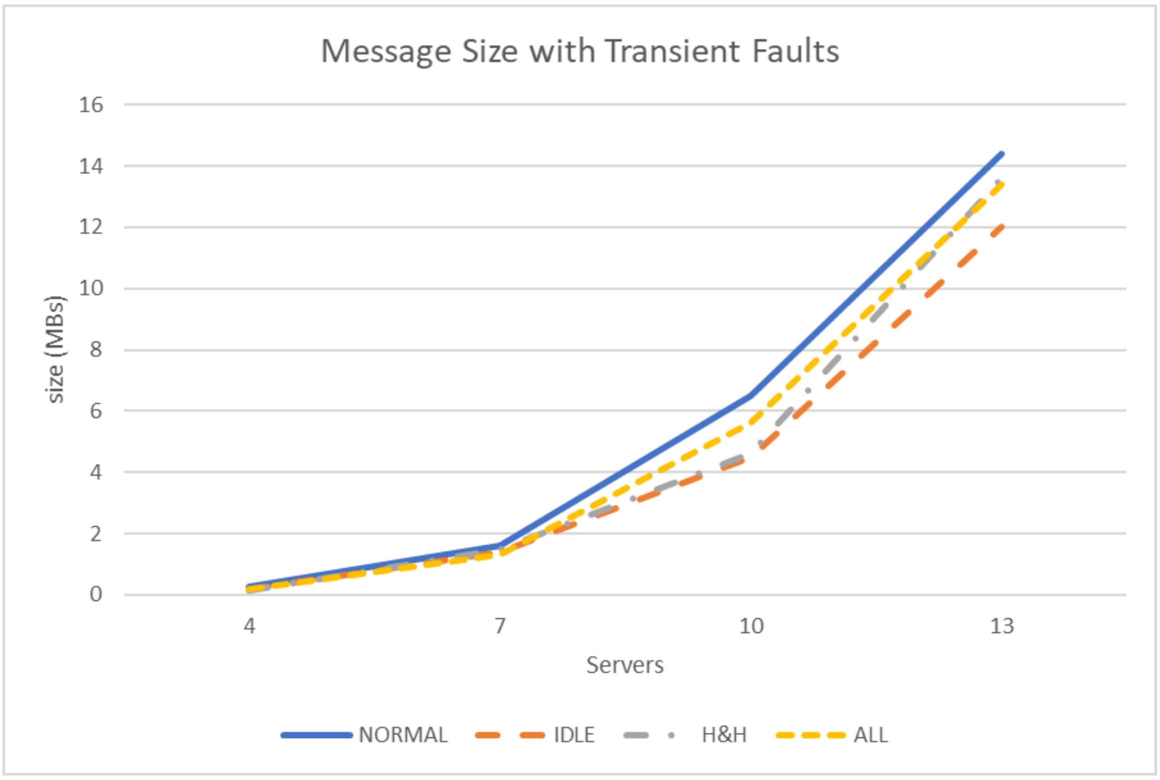


Figure 5.12 Message Size – constant clients and increasing servers with transient faults

Lastly, regarding message size, an increase is indicated in Figures 5.11 and 5.12 as more servers are added to the system. Message size is affected by the number of servers and the size of their proposal, which is stored in each process' *msg* array. Transient faults alter the stored messages to a one-byte value, while Byzantines transmit one-byte messages.

Size soars with more servers as a virtue of having more proposals saved at any given time. As expected, the lowest average message size is when faulty nodes remain idle. This means that they do not send values to other servers, thus correct nodes only store suggestions from non-faulty servers. This makes the *msg[i]* of each process p_i smaller due to a *n-f* upper bound of maximum number of messages contained. On the other hand, in faulty clean experiments we have the most MBs required for the exchanged messages. This is anticipated as the message of a correct server is larger than the values proposed by Byzantine processes. Therefore, in *HALF and HALF* and *ALL* unit tests, where *f* values of a full *msg[i]* are Byzantine proposals, the message size is similar as indicated in their graphs. Their size is moderately smaller than the size in *NORMAL* scenarios, too.

The same growth trend is shown in both transient environments and transient free ones. The differences among the experiments where Byzantines send values are almost negligible. Nevertheless, messages in transient environments are slightly smaller in comparison with the non-transient systems. This is the case for any scenario that has taken place. This is a completely normal observation due to transient value altering. When messages are altered, their size is equal to one byte as the only goal of the transient fault in our environment is the confusion of correct processes. Therefore, until the fault gets cleared in every process, *msg[i]'s* messages are gradually compacter. These broadcasts of smaller values lower the average message size measured.

5.4.3 Convergence Time

The bar chart in Figure 5.13 indicates the convergence time, as defined previously, for several execution scenarios. These scenarios contain a constant number of clients, 25, with an increasing number of servers, i.e., 4, 7, and 10. Additionally, every unit test was assessed with two versions of transient appearance probability, namely 0.5 and 1. The transient fault considered in these experiments is the arbitrary state initialization of a server. That is, servers in this arbitrary state, launch the protocol with messages in *msg*, which include faulty values for every process in the system. These faulty messages hold similar content for every process, namely the byte of "0" character as value and a constant *num* number. All the types of messages are created, i.e., *init*, *echo*, and *ready* messages for each system node. Thus, the faulty messages with the same sender support each other's existence. Consequently, convergence time is measured as the time between the beginning of the algorithm until its first delivery of a correct message. Descriptions used in x-axis specify the behavior of Byzantine processes, as presented previously.

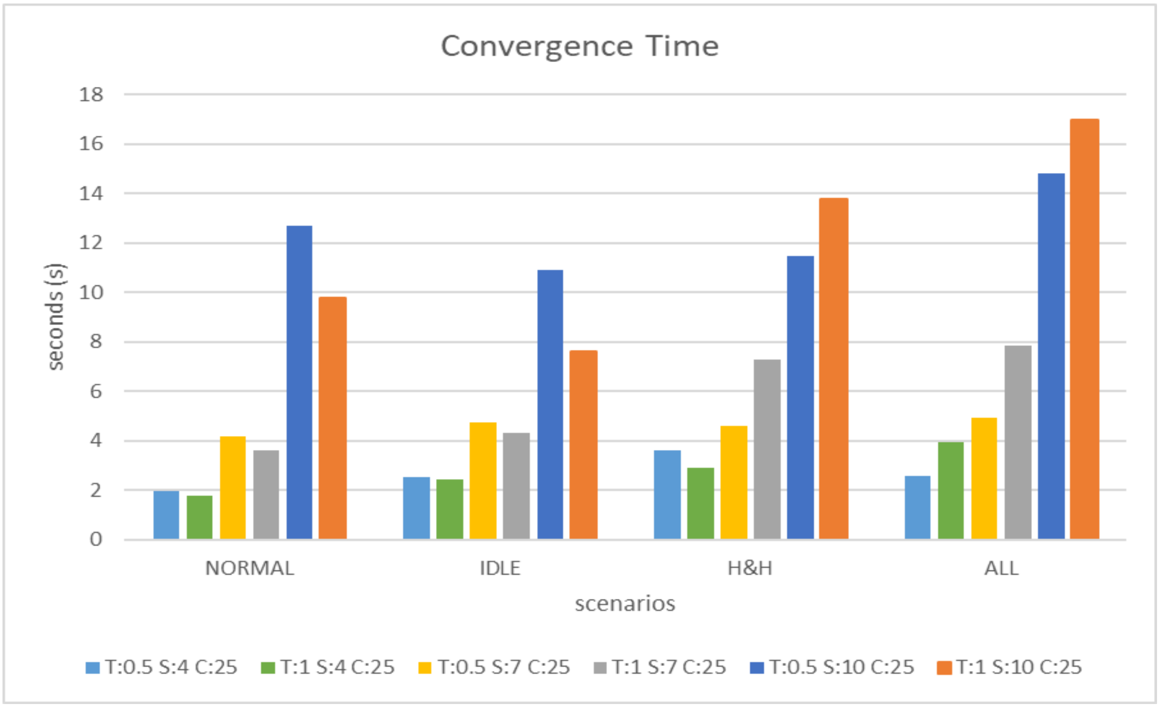


Figure 5.13 Convergence Time graph

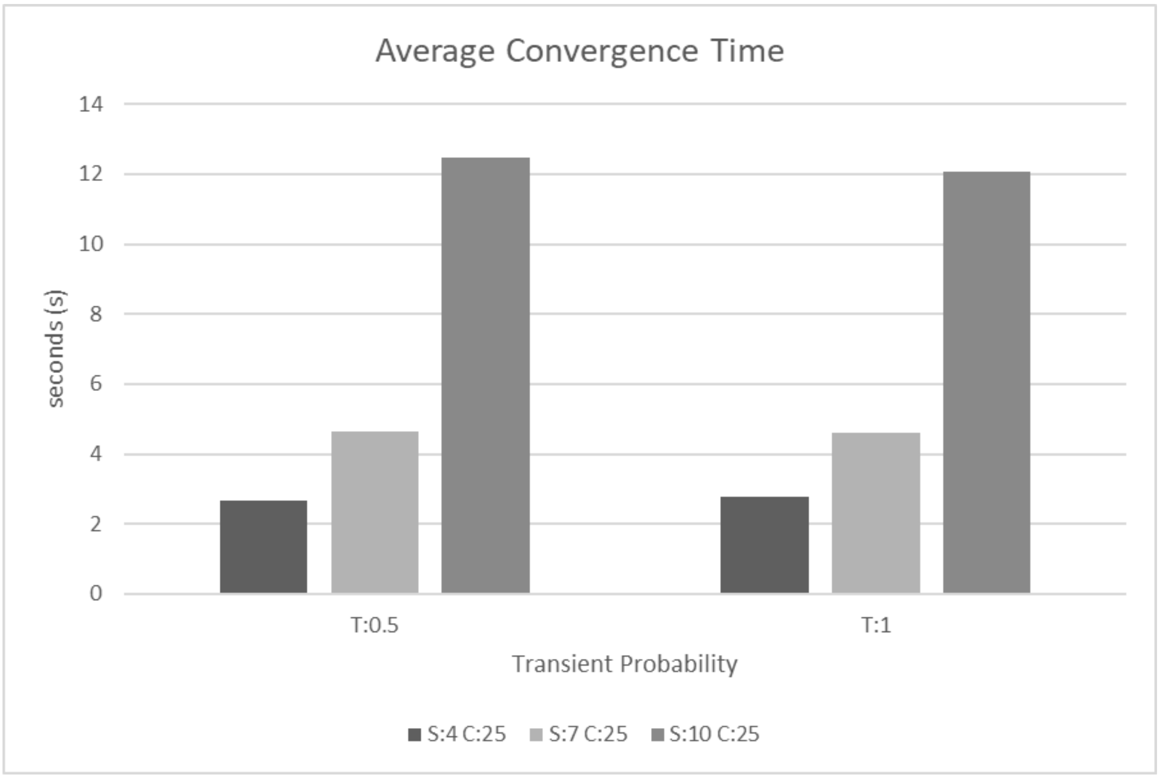


Figure 5.14 Average Convergence Time graph with different transient probabilities

As it is shown in Figure 5.13, the more servers in the system the more time is needed for convergence. This is expected because of the time needed to check for messages in *msg*, which is in the order of $O(n^3)$ as mentioned above, as well as the time required for a server to get messages from every affected node and realize that messages contained in its *msg* array are incorrect. In same Byzantine behavior scenarios, the relationship between the two measurements with the same servers is the same regardless of system size. Figure 5.14 indicates that transient probability does not affect the average convergence time of the experiments. This average is calculated by considering all the different Byzantine behaviors of each system. Nevertheless, larger systems require more time to reach a safe state.

Furthermore, when Byzantine processes send malicious values, a greater transient probability indicates an increase in convergence time, whereas in correct environments or in crash failure scenarios the opposite is the case. This is expected due to the dichotomy of the transient affected and transient free servers in *NORMAL* and *IDLE* scenarios, where malicious values are not transmitted, when transient probability is equal to 0.5. Faulty messages supplement each other's existence so when every process is affected and no malicious values are sent, servers are able to detect the transient fault more easily as they check the messages that are stored in their *msg* array and detect the presence of messages that were not broadcast by them. In tests where only half of the servers are affected, it is more difficult for them to detect the transient fault as they have stored messages from nodes that did not actually send anything. The unaffected processes have a clear *msg* array in their memory initially, so they do not detect any fault in their state. Hence, transient free servers do not send any message with contents capable to indicate to others that a transient fault has altered values in their memory. This would happen when they get their first request. Therefore, the time required for servers to realize that a transient fault has occurred when only half of them are affected is greater than the time needed when all of them have altered values in *msg*.

The largest system needs the least amount of time to converge when there are crash failures in the system. The main reason behind this is the fewer messages contained in *msg* array, which decrease the time needed for several algorithm's evaluations and flushes to take place. This is not shown in smaller systems as a virtue of the small difference of the affected servers among the two cases of the same configuration. They achieve their fastest time in environments with no presence of Byzantine faults. The longest time is needed when Byzantine malicious messages are sent, which support transient faults value-wise, regardless of the number of servers. The ten-process system has a moderate time difference between 0.5 and 1 probability values in the first two scenarios. This difference diminishes as Byzantines start to send inappropriate values. In any other experiment, the pattern followed is the exact opposite as malicious messages worsen the time gap between identically sized systems.

Overall, the general structure of the measurements on each Byzantine behavior separately is identical among the different number of servers. The size of the system affects the average convergence time, whereas transient probability has less influence on the latency to achieve a safe state.

5.5 Comparison with Existing Work

Our protocols are compared with the experimental measurements of the algorithms of Correia et al. [15]. The usage of these algorithms as a base for the newly designed ones implies the essence of these comparisons. This indicates the exact difference created by self-stabilization as the two atomic broadcast algorithms have a common origin. The evaluation of the studied algorithms and its non-self-stabilizing counterparts are in terms of the main three metrics mentioned above. Namely, operation time, average message complexity, and average message size. Because of the non-transient fault tolerance nature of the Correia et al. s' [15] protocols, the comparisons made refer to the scalability of either clients' or servers' aspect only. The systems used for the measurements of both protocol stacks are identical as the studied algorithms have been developed on top of the work done in a previous thesis [35]. That is, the graphs presented below are the ones shown in that diploma project. Our testing scenarios take into consideration the coordinated attack of the Byzantine processes, i.e., *ALL*, which send the same value to every node, whereas the graphs in the previous thesis [35] do not contain measurements regarding this unit test. Therefore, the comparisons were made based on the common scenarios, which are defined by the behavior of the Byzantine processes and the number of servers and clients used. It is critical to be mentioned that our experiments took place in a five-machine cluster, whereas the tests of the original ABC stack were done in a nine-machine one. Additionally, an important thing to consider is that ABC's message metrics take into consideration the faulty processes, whereas our graphs are covering up only correct servers.

5.5.1 Clients Scalability Comparison

The non-self-stabilizing atomic broadcast graphs of operation time, message complexity, and message size, with constant severs, are presented in Figures 5.15, 5.16, and 5.17, respectively. The tests executed for gathering the data of the graphs below kept the number of servers steady and equal to four, just like our experiments. Comparisons include only the transient clear tests to be able to evaluate similar context metrics. These are presented in Figures 5.1, 5.3, and 5.5, which correspond to operation time, message complexity, and message size respectively.

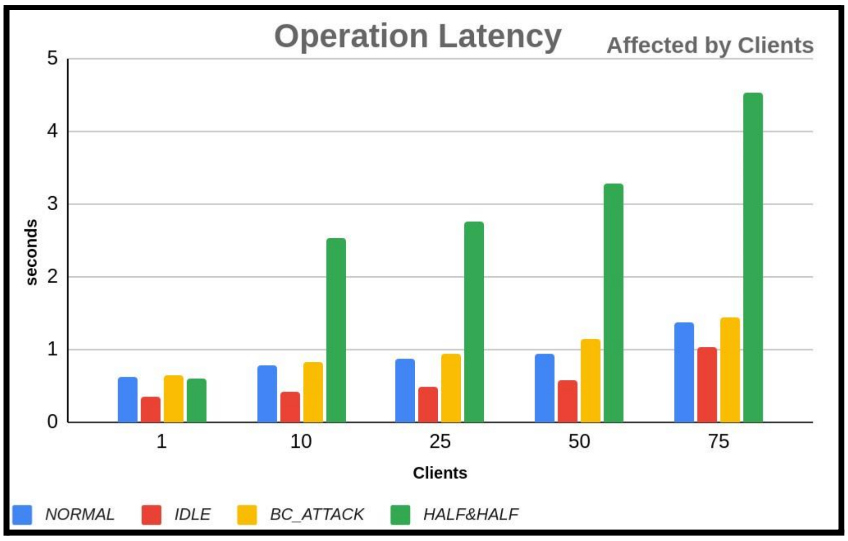


Figure 5.15 Operation Latency affected by the increase of clients as illustrated in [35, Fig. 5.4]

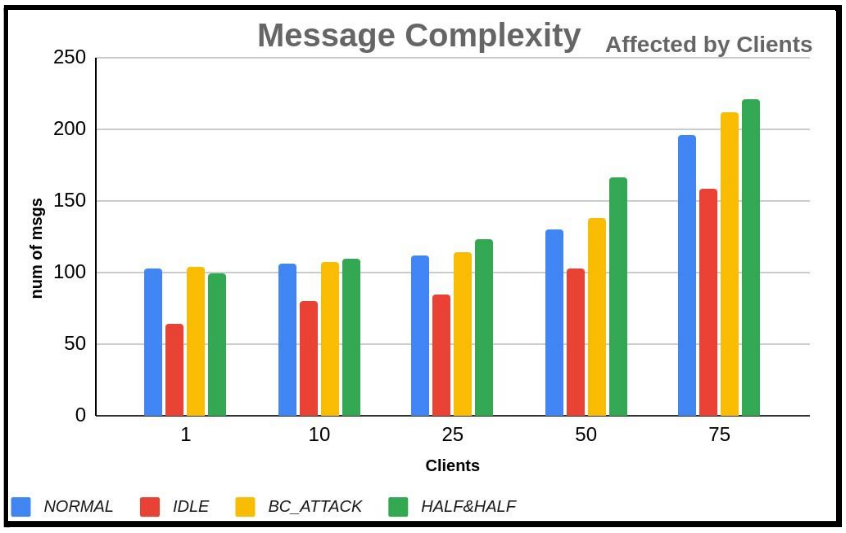


Figure 5.16 Message Complexity affected by the increase of clients as illustrated in [35, Fig. 5.5]

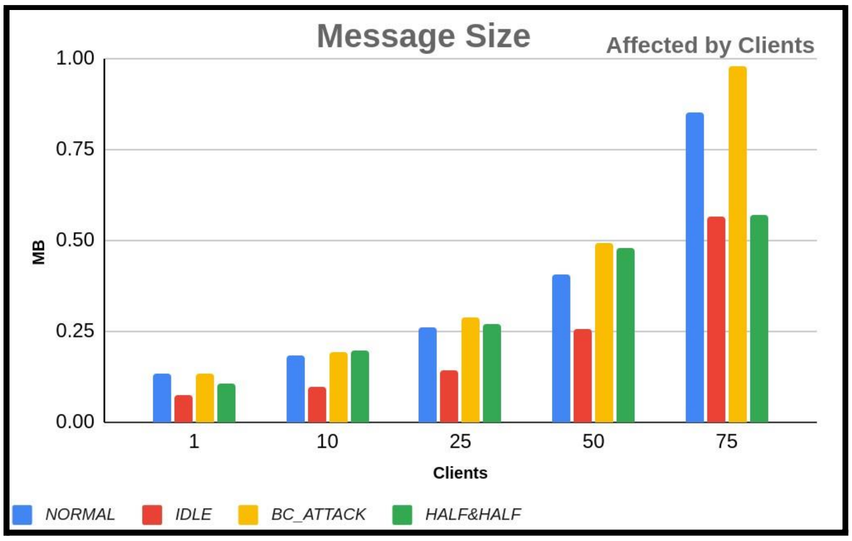


Figure 5.17 Message Size affected by the increase of clients as illustrated in [35, Fig. 5.6]

To start with, operation time is significantly lower in original atomic broadcast compared to its self-stabilizing version. This is expected because to achieve self-stabilization, additional assessments on saved values are made, which increase the overall execution time. To elaborate more, the shape of the graphs is different as ABC's time is increasing in the same way, whereas SSABC lowers the climb in fifty client environments. The studied protocol is more stable among the different Byzantine behaviors, while the original atomic broadcast algorithm needs substantially more time when Byzantines send malicious values, i.e., *HALF and HALF* scenario. In detail, the time needed in ABC for *NORMAL* and *IDLE* experiments is less than a second even with fifty clients. In contrast, the studied protocol requires more than 10 seconds to deliver requests from fifty clients but only three times the ABC latency to atomically broadcast them in *HALF and HALF* tests. These differences are anticipated because each server that uses the self-stabilizing algorithms can only manage at most one request per time. Additionally, more messages are sent over the network, which imply extra transmissions overhead and supplementary time to manage.

Furthermore, the message complexity graph shape of the self-stabilizing stack is like its original counterpart. Messages sent increase in a linear fashion as more clients are added to the system. Nevertheless, the studied algorithm sends more values until the end of its operation. This is due to the periodic broadcasts that are needed to circumvent the problems of transient faults. More proposal rounds because of the limited suggestion capabilities of servers result in more iterations of the main *do-forever* body, which means more messages are sent. In addition, each process broadcasts its values while waiting to get the consensus result of the beneath layer. All these extra exchanges make the message complexity slightly larger in the studied protocol with even 1.5 times more messages in *NORMAL* tests. However, the transmission of multiple *echo* and *ready* messages in one broadcast, instead of a separate one for each message, drops the difference of the two protocols. Moreover, in ABC, messages exchanged are relatively stable among the various Byzantine behaviors by considering each system instance separately. In contrast, the studied algorithm sends an additional considerable number of messages in fault free scenarios.

Lastly, original atomic broadcast's message size rises with more clients, while the studied protocol's size is steady along several tests. As mentioned previously, our measurements consider the size of correct servers only, which explains the overall stability of our graph in Figure 5.5. In the older thesis, the smaller size in *IDLE* experiments is due to the consideration of the crashed processes in the average calculation. All the other scenarios have a similar size estimation. The most notable difference, however, lies in the actual amount of MBs required for each message. Our protocol needs more memory for each message in environments with 1 or 10 clients. This is the case as a virtue of the storage of every server proposal and the broadcast of a *msg* entry which contains them. In addition, each process broadcasts all its messages every time. On the contrary, ABC's message size in systems with twenty-five or more clients is greater than the average size of the studied protocol. This is explained by the larger vector proposals of the non-self-stabilizing atomic broadcast. Our algorithm can suggest at most an *n*-sized vector to SSVC, whereas in the normal ABC servers do not have a limit on the suggestion. As more requests are received, due to the climb of clients, servers propose larger vectors to VC, which affect the average message size in an upwards manner.

5.5.2 Servers Scalability Comparison

The non-self-stabilizing atomic broadcast graphs of operation time, message complexity, and message size of constant client tests are presented in Figures 5.18, 5.19, and 5.20, respectively. These experiments used a constant number of clients, namely twenty-five, similar to our servers' scalability scenarios. For better comparisons, we only consider our transient free unit tests. These metrics are indicated in Figures 5.7, 5.9, and 5.11, which correspond to operation time, message complexity, and message size, respectively.

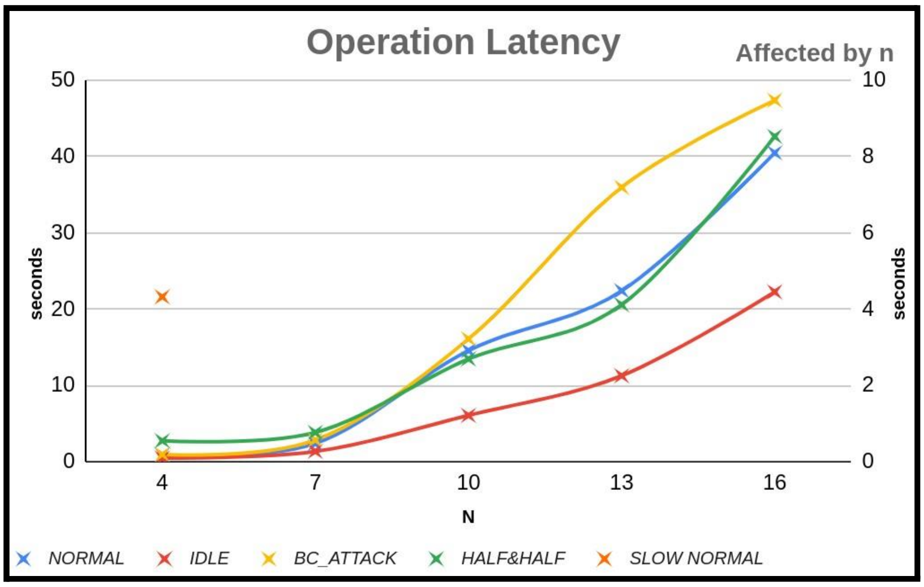


Figure 5.18 Operation Latency affected by the increase of servers as illustrated in [35, Fig. 5.7]

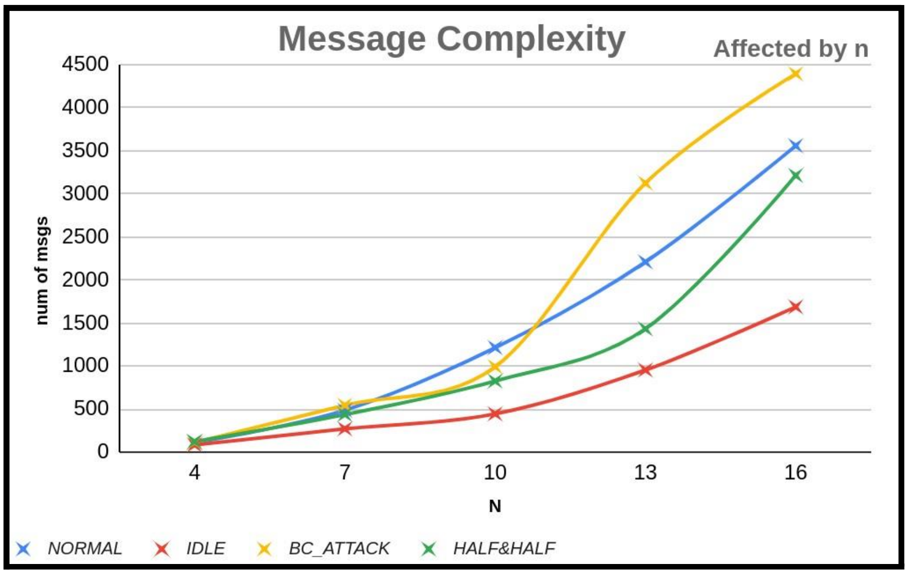


Figure 5.19 Message Complexity affected by the increase of servers as illustrated in [35, Fig. 5.8]

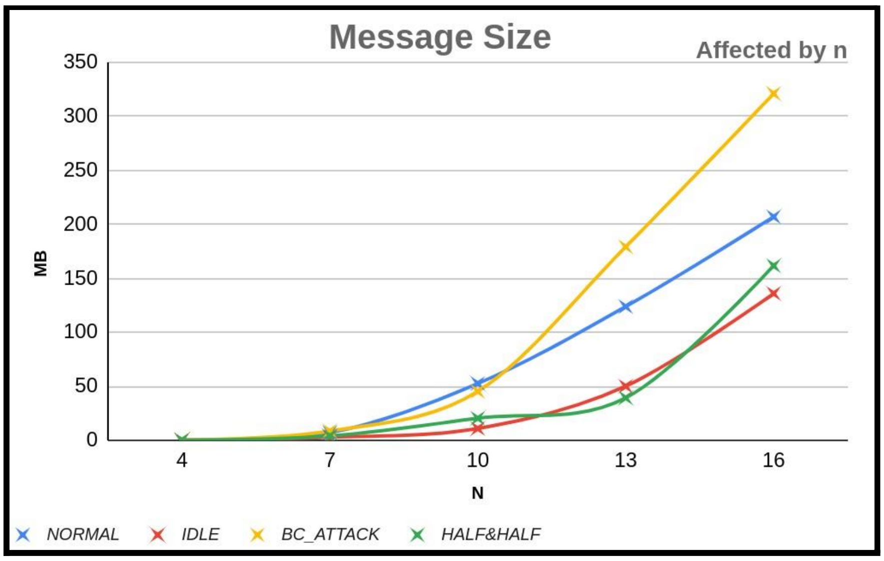


Figure 5.20 Message Size affected by the increase of servers as illustrated in [35, Fig. 5.9]

To begin with, the operation latency of both protocols is vastly affected by the number of servers in the system. While time increases in both protocol stacks, the studied algorithm's rising trend slightly decreases in larger systems. Crash prone environments have the lowest operation time both in our work and in ABC. The maximum time in our system, however, is achieved when Byzantine nodes send malicious values, whereas in the older thesis' experiments the *NORMAL* scenarios take the most time. These differences could be explained by the hardware setup used for the experiments. Similarly with client scalability, the absolute time variation is the major distinction of these graphs. As expected, the studied protocol needs significantly more time to deliver every request due to the enforced proposal number limitation. Also, the additional evaluations of the *msg* array to track transient faults cumulatively contribute to operation time.

Moreover, the amount of the exchanged messages surges with the addition of more servers. This is due to the necessity of broadcasting values to more recipients for the correct operation of the algorithms. The actual messages sent in fault-free environments is similar to both original and self-stabilizing protocols. A larger variation exists among ABC's messages in different Byzantine behaviors, whereas the studied protocol is more stable with malicious actions. Besides this, the *IDLE* test of the original algorithm stack exchanges the least number of messages, while our *IDLE* scenarios are approximately the same as the other experiments. This is due to the consideration of crashed processes in the ABC's average message complexity metric, which plunges the actual average value calculated.

Finally, the message size increases in both protocol stacks. The reason behind this growth is different among the two atomic broadcast algorithms. While the climb of the studied protocol is because of the more proposals stored in *msg* entries, the rise of the original ABC is due to the greater size of the suggestions to lower levels. The different causes affect the MBs needed for the average message. The self-stabilizing algorithm has a growth which is expected with the linear addition of servers because of the analogous increase of the proposals in *msg*. Conversely, the original protocol stack's size soars in a faster rate. Servers propose larger vectors, as a virtue of constructing a vector with all the reliably delivered items, thus, the messages to transmit them are substantially larger. As more nodes do the same, the average size of the messages exchanged moves towards the size of the proposed vectors, which is proportionate to clients' requests instead of system's servers, which is the case in SSABC.

5.6 Experimental Summary

In general, the measurements that took place indicate the expected behavior of the studied algorithm. None of the metrics is vastly affected by the presence of either Byzantine or transient faults. This is crucial to present the stability of the protocol and the fact that the extra messages and time overhead required to clear a transient fault is almost negligible compared to its transient free execution. Byzantine malicious actions produce only minor variations in the evaluated metrics and crash failures are managed more easily in the studied protocol. Despite the difference between the utilized machines in our cluster and the one used in the original ABC evaluations; significant conclusions were made. The time-wise scalability of the self-stabilizing atomic broadcast is worse compared to the original. This is anticipated as more checks have been placed in the algorithm's main body to assess whether a transient fault has occurred. Additionally, the enforced constraint of at most one request proposal leads to this difference between the studied protocol and the non-self-stabilizing one. Nevertheless, SSABC needs smaller messages to be transmitted, which depletes the network's bandwidth and lowers the transmission time. Moreover, only a few extra messages are required to enforce self-stabilization due to the forward of many *echo* or *ready* messages in one broadcast instead of separate ones. The last observations indicate that message-wise, the benefits of transient tolerance certainly circumvent the drawbacks of its absence. Finally, the convergence time is approximately the same regardless of the amount of the affected processes, but systems with more servers require more time to achieve a safe state.

Chapter 6

Conclusion

6.1 Summary

The main objective of this thesis was the validation and experimental evaluation of self-stabilizing versions of vector consensus and atomic broadcast. The algorithms proposed by Correia et al. [15] were used for the self-stabilizing transformation. After the study of the protocols, a development phase took place using the de facto language for distributed systems, Go, and a state-of-the-art communication library, namely ZeroMQ. The implementation of the algorithms was used to assess their behavior in various scenarios, while performing an experimental evaluation. These unit tests were used to evaluate algorithms' correctness and for measurements that had to do with the number of messages sent, message size, operation latency, and their convergence time. A general performance profile of the algorithms was concluded, based on their numerical assessments. Comparisons between the studied protocols and their non-self-stabilizing counterparts took place to estimate the influence of the self-stabilizing changes. Consequently, the creation of a transient tolerant Atomic Broadcast algorithm is demonstrated to be possible as a virtue of the present study.

6.2 Challenges

Some challenges of this thesis regard the transformation of the pseudo-code to Go code. This was an issue because Go does not support set types and the *map* variable used for storing messages could not take our message tuple structs as keys. Additionally, Go does not support functional programming despite having a lot of features where functional principles are applied. The benefits of Go language, however, circumvent the impact of its drawbacks so we proceeded with its usage. Moreover, the limitation of the cluster resources provided another obstacle for this project. The number of machines in the cluster is bounded to 5, while many of them are low spec machines. This made the scalability of the results difficult as experiments with greater number of servers and clients could not be tested properly. Despite that, the results of our measurements are representative enough for the amount of processes considered in the system. Therefore, the effectiveness of the self-stabilizing protocols could be assessed and comparisons with other algorithms could be performed.

6.3 Future Work

In the future, some optimizations of the studied protocols could take place. At first, an effort to lower the time complexity of the *do-forever* loop as this will help dramatically the operation latency. This could be done by either using better data structures for storing messages, like trees with logarithmic depth, or by handling differently the actions done on messages. Message broadcasting could occur periodically and not in every iteration of the *do-forever* loop to lower the number of messages sent over the network and the time to process them on the recipient's side. Moreover, the studied atomic broadcast algorithm could be altered for servers to be able to propose multiple requests at a time. This introduces a complexity on its proof of correctness and additional self-stabilization properties guarantees. However, this would certainly decrease the lower layer's function calls, thus plunging the overall operation time. Furthermore, the same experiments could take place but, in our cluster's full capacity by using all nine machines for better results through finer evaluation of the protocol's scalability with more servers and more clients. Another suggestion is to test the system in a more geographically distributed processes' network or in higher performance machines to gain a better sense of its real-life applicability. Finally, a full self-stabilizing stack could be implemented, with self-stabilizing binary and multivalued consensus modules. This would provide more accurate comparisons between transient tolerant and other ABC stacks. This would be a more representative self-stabilizing implementation and would help to assess transient tolerant behaviors of algorithms in real-world simulations.

Bibliography

[1] Luciana Arantes, Roy Friedman, Olivier Marin, and Pierre Sens. Probabilistic byzantine tolerance for cloud computing. In 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS). IEEE, sep 2015.

[2] Luciana Arantes, Roy Friedman, Olivier Marin, and Pierre Sens. Probabilistic byzantine tolerance scheduling in hybrid cloud environments. In Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17. ACM Press, 2017.

[3] Anish Arora and Mohamed G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. IEEE Transactions on Software Engineering, 19(11):1015–1027.

[4] Atomic Broadcast and Self-stabilizing Atomic Broadcast Stacks. https://github.com/evangeloug/Self-stabilizing-Atomic-Broadcast, (Accessed: 2022-05-17).

[5] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983, pages 27–30, 1983.

[6] Alexander Binun, Thierry Coupaye, Shlomi Dolev, Mohammed Kassi-Lahlou, Marc Lacoste, Alex Palesandro, Reuven Yagel, and Leonid Yankulin. self-stabilizing byzantine-tolerant distributed replicated state machine. In stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings, pages 36–53, 2016.

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

[8] Ethan Buchman, Tendermint: Byzantine fault tolerance in the Age of Blockchains, Thesis Project, University of Guelph, (2016).

[9] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In 2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings, pages 167–176. IEEE Computer Society, 2002.

[10] Christian Cachin. State machine replication with byzantine faults. In Charron-Bost et al. [39], pages 169–184.

[11] Cantor's Pairing Function. https://mathworld.wolfram.com/PairingFunction.html, (Accessed: 2022-05-17).

[12] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, pages 173–186, 1999.

[13] Tushar Deepak Chandra and Sam Toueg. "Unreliable failure detectors for reliable distributed systems". In: Journal of the ACM (JACM) 43.2 (1996), pp. 225–267.

[14] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solvingconsensus.Journal of the ACM, 43(4):685–722, 1996.

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

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

[17] Shlomi Dolev and Jennifer L. Welch. self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM, 51(5):780–799, 2004.

[18] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. 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, pages 84–100, 2018.

[19] Shlomi Dolev. self-stabilization. MIT Press, 2000.

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

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

[22] ESS – BFT Efficient self-stabilizing Byzantine fault tolerance. https://www.cs.ucy.ac.cy/essbft/, (Accessed: 2022-05-11).

[23] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: Journal of the ACM (JACM) 32.2 (1985), pp. 374–382.

[24] Golang Gob. https://golang.org/pkg/encoding/gob/, (Accessed: 2022-05-17).

[25] Jim Gray et al. "The transaction concept: Virtues and limitations". In: VLDB. Vol. 81. 1981, pp. 144–154.

[26] Ted Herman. Probabilistic self-stabilization. Inf. Process. Lett., 35(2):63–67, 1990.

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

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

[29] Message Brokers. https://www.ibm.com/cloud/learn/message-brokers, (Accessed: 2022-05-24).

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

[31] Zarko Milosevic, Martin Hutle, and Andr´e Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011),Madrid, Spain, October 4-7, 2011, pages 235–244. IEEE Computer Society, 2011.

[32] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Ver´ıssimo. RITAS: services for randomized intrusion tolerance. IEEE Trans. Dependable Secur. Comput., 8(1):122–136, 2011.

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

[34] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching agreement in the presence of faults”. In: Journal of the ACM (JACM) 27.2 (1980), pp. 228–234.

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

[36] Michael O. Rabin. Randomized byzantine generals. In 24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983, pages 403–409. IEEE Computer Society, 1983.

[37] Marco Schneider, self-stabilization, ACM Computing Surveys, Vol. 25, No. 1, 1993.

[38] Andrew S. Tanenbaum and Maarten Van Steen. Distributed Systems. Pearson Education, 2013.

[39] The Go Programming Language. https://golang.org/, (Accessed: 2022-05-17).

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

[41] ZeroMQ Socket API. https://zeromq.org/socket-api/, (Accessed: 2022-05-24).

[42] ZeroMQ. https://zeromq.org/, (Accessed: 2022-05-17).

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