CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Implementing Self-stabilizing Byzantine Fault-tolerant State-machine Replication

a Proof of Concept, Validation and Preliminary Evaluation

Master's thesis in Computer science and engineering

AXEL NIKLASSON
THERESE PETERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Implementing Self-stabilizing Byzantine Fault-tolerant State-machine Replication

a Proof of Concept, Validation and Preliminary Evaluation

AXEL NIKLASSON
THERESE PETERSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Implementing Self-stabilizing Byzantine Fault-tolerant State-machine Replication: a Proof of Concept, Validation and Preliminary Evaluation
AXEL NIKLASSON
THERESE PETERSSON

Supervisor: Elad Michael Schiller, Department of Computer Science and Engineering
External supervisor: Chryssis Georgiou, University of Cyprus, Department of Computer Science
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Execution times of client requests for a state machine with a fixed size deployed to six machines with and without the self-stabilizing property. The execution times varies between two and five seconds and the overhead is relatively constant.

Typeset in LaTeX
Gothenburg, Sweden 2019

Implementing Self-stabilizing Byzantine Fault-tolerant State-machine Replication: a Proof of Concept, Validation and Preliminary Evaluation

AXEL NIKLASSON
THERESE PETERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As the usage of cloud-based services increases, so does the demand on high availability and reliability on these services, which can be achieved using state machine replication. This project covers the implementation of a proof-of-concept, validation and preliminary evaluation of the Self-Stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors algorithm presented by Dolev, Georgiou, Marcoullis and Schiller. We are the first to implement and practically validate the algorithm, along with performing a preliminary evaluation using the PlanetLab EU platform. The implemented codebase along with its extensive tooling can be used in other projects implementing and evaluating distributed algorithms and at the time of writing, the codebase is a fundamental part of two other projects at the department.

The main goal of this project is to perform a practical validation of the algorithm, which has successfully shown that the algorithm performs correctly for all implemented test cases. The primary implementation of the algorithm is implemented as a list supporting only append operations, similar to a blockchain. The second implementation removes all overhead related to an ever-growing list and only supports a ”no operation” operation. The results from the preliminary evaluation, conducted with both implementations, indicate that the overhead introduced by self-stabilization increases as the number of nodes increases and is also depending on the chosen state machine. It has been shown that the system performs best with a minimum (six) number of nodes. Furthermore, it has also been shown that such a system is able to converge to a serviceable system state within seconds, which compared to human intervention can be considered to be very satisfactory.

Keywords: Computer science, engineering, self-stabilization, Byzantine fault-tolerance, failure detection, replicated state machine, cloud storage

Acknowledgements

This project was jointly supervised by Chalmers University of Technology and University of Cyprus. The supervision team from Chalmers included Elad Michael Schiller while the supervision in Cyprus included Chryssis Georgiou and Ioannis Marcoullis.

We want to thank our supervisor Elad Michael Schiller for guiding us through the self-stabilizing jungle with his expertise and working closely and remotely with us throughout the project, making it possible for us to combine our passions for both travelling and computer science.

We also want to thank Chryssis Georgiou and Ioannis Marcoullis for helping us throughout the whole project and for happily welcoming us to Cyprus in the best possible way. Working with you in Cyprus was critical for the success of this thesis and something we warmly look back on.

We would like to thank the Erasmus organisation at Chalmers and in Cyprus for the scholarship enabling us to go to Cyprus.

Lastly we want to thank the bus driver of route 259 in Cyprus, for giving us energy in the morning with his “Kalimera”-greetings and waving.

Axel Niklasson & Therese Petersson, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1
Introduction

As the usage of cloud-based services increases, so does the demand on high availability and reliability on these services. A few examples of such services are those that handle distributed computational tasks such as blockchains or distributed storage solutions such as Google Drive[1] and Dropbox[2]. Many of these cloud-based services distribute their data across several machines in order to satisfy the high reliance demand, possibly by implementing a replicated state machine which stores the data using multiple physical servers, removing the single point of failure. However, failures do occur and cloud-based services need to tolerate a broad set of failures.

An example of a failure which could be named *Byzantine*, is if one or several machines (processors) are acting arbitrarily, meaning that it is not easy to spot the incorrectness, according to the protocol, of the machine(s) [1]. This uncertainty in knowing which processors are behaving correctly can have big consequences. A machine can intentionally behave maliciously in order to try to corrupt the other machines and would therefore too be classified as Byzantine. Other failures, which are non-malicious but could still create similar results, are software bugs and operator mistakes. The processors with a Byzantine behaviour can also simulate asynchronous behaviour, by delaying messages, as well as asymmetric failures in which the Byzantine processors send incoherent messages.

A *Byzantine fault-tolerant* (BFT) system is challenging to design but highly important to achieve. An even stronger fault-tolerant requirement can be achieved by *self-stabilization*. Self-stabilizing systems can start in an arbitrary state and converge to a legal execution, i.e. in a state satisfying the system requirements [2]. This property is desirable when a system should be operational in the presence of failures. It also means that in case of a failure forcing the system to a illegitimate behaviour, the system can automatically recover from it.

The benefits achieved by the self-stabilization property in combination with Byzantine fault-tolerance, can save both time and manpower, which could lead to more economical and productive systems. Engineers do not need to focus on monitoring and managing the running systems, which is desirable for production systems. In this project, a self-stabilizing Byzantine fault-tolerant system is implemented and validated.

[1] https://www.google.com/drive/
[2] https://www.dropbox.com

1.1 Background

Cloud-based services, such as storage services, are becoming more popular and as a result, the demand for resilient and highly available services increases. Most users expect to be able to access their files at all times and therefore, the underlying systems providing these services need to provide high availability while also being reliable. Should they fail to do so, users may switch to a competing service.

A Byzantine fault-tolerant system provides a high resilience to failures, which is desirable for any distributed system. The first Byzantine fault-tolerant algorithm, developed for a replicated state machine in an asynchronous system such as the Internet, is the so called *Practical Byzantine Fault-Tolerant (PBFT)* by Castro and Liskov [1]. Their paper shows the overhead introduced when introducing the Byzantine fault-tolerance property to a file system protocol. The result indicates that the actual overhead is relatively small, but their solution assumes that the system starts in a specific setting. Therefore, the system might, due to for example stale information, end up in a situation with processors having invalid states and human intervention is necessary to make the system functioning correctly again.

Cloud-based services need to consider a fault model including for example Byzantine faults along with a complimentary set of transient faults which could lead to stale information. These transient faults violate the assumptions needed to reach a legal execution of the system. When such a violation occurs, once the last transient fault occur, it is hard to reason about the state of the system. *Self-stabilizing* systems consider an arbitrary starting state, from which the system returns to a legal execution within a bounded number of execution steps; no human intervention required.

In [3], Dolev, Georgiou, Marcoullis and Schiller present the first self-stabilizing Byzantine fault-tolerant state machine replication service based on failure detectors. This project is the first to implement and validate the algorithm by Dolev *et al.*, further referred to as the SSPBFT algorithm. In order to examine the performance of the implemented algorithm, a preliminary evaluation was also performed.

1.2 Purpose

The main purpose of this project is to validate the correctness proof of the SSPBFT algorithm by Dolev *et al.* [3]. In order to do so, we aim at producing a software implementation that contains truthful code with respect to the implemented algorithm, meaning that it models the algorithm as correctly as possible. We also aim at implementing tests which can practically validate the correctness of the algorithm based on invariants found in the proof of [3]. A preliminary evaluation of the implemented system will also be conducted, providing insights of the system performance.

1.3 Limitations

We note that the self-stabilizing algorithm by Dolev *et al.* is written in an event-driven manner [3]. That is, every procedure is raised as a result of a timer expiration or a message arrival. This programming style is known for its execution efficiency and ability to scale well with respect to the number of connections. The implementation carried out in this project has not been conducted in an event-driven manner due to its complexity and using other, more familiar, implementation approaches was crucial to enable the rapid development needed in this project.

The main objective of this project is to validate the correctness proof of the algorithm. We have decided to use a rapid development approach that allowed developing the non-trivial algorithm by Dolev *et al.* quickly and then focusing on making sure that the proof invariants indeed hold [3]. Due to this, we have decided to use a thread-based approach when designing and implementing the application. Threads will be used for receiving messages both over TCP and UDP, as well as for sending all messages over UDP. There is an event-driven implementation packaged along with the library used for the TCP communication, which will be used for the sender channels communicating over TCP rather than threads in an effort to reduce the overall number of threads. Moreover, each algorithmic module uses one or two threads each to execute the algorithmic logic and there are also some other threads in usage by the system, for various other types of functionality such as, for example, a Web API. A summary of all threads used can be found in Table A.1 in Appendix A, which shows that the total number of threads used for one node, T, can be expressed as $T = n + 9 + \lambda$, where n is the number of nodes in the system and λ is the number of currently running UDP Sender timeout threads. Consequently, the number of threads active for each node in a system of 12 nodes is at least 21 threads and often more, depending on the amount of timeout threads running at the time.

We note that existence of better implementation approaches such as epoll[3], asynchronous I/Os and Data Plane Development Kit[4] are known to be more suitable alternatives for implementing prototypes with respect to communication performance. Measurements during evaluation indicate that the number of threads resulted in a severe impact on the performance, leading to CPU thrashing and slowing down the performance a great deal. Nevertheless, our rapid prototype approach was indeed beneficial since we needed to devote time to both development and validation including the preliminary evaluation. We feel that our rapidly developed prototype can pave the way for a "full-blown" implementation.

1.4 Related work

The concept of a replicated state machine is a general approach to achieve fault-tolerance in distributed systems. In this section, we focus on the work related to

[3]http://man7.org/linux/man-pages/man7/epoll.7.html
[4]https://www.dpdk.org/

Byzantine fault-tolerant (BFT), replicated state machines and work related to self-stabilization is also presented.

As mentioned, Castro and Liskov developed PBFT, the first BFT algorithm developed for a replicated state machine applicable to asynchronous systems [1]. The algorithm provides safety without assuming synchrony in the system, a property necessary in order to be deployed in real-world like systems. It is fairly easy for an adversary to delay messages in distributed systems, and if there is a bound on the delivery time of a message (i.e. synchronising the system for safety reasons) this might jeopardise the correctness of the system. One core part of the PBFT algorithm is the usage of a *primary* processor, whose job is to conduct the ordering of incoming requests from the clients. The implementation is presented as a generic library to be used for Byzantine fault-tolerant (BFT) systems. The algorithm is however not self-stabilizing since it requires to be repaired via external (human) intervention for certain failures. The work by Castro and Liskov paved the way for other BFT algorithms, inspired by the approach taken in PBFT.

An algorithm called *Zyzzyva* by Kotla *et al.* aimed (and managed) to reduce the overhead of using a BFT-protocol by adapting a speculation function [4]. This function allows the processors to directly act upon a client request without waiting for the agreement protocol to finish. This means that a processor accepts the ordering of the primary and applies the request, including sending a reply back to the client, without consulting the other processors. This, of course, has consequences; one being the possibility of an inconsistent state among the processors if the primary is faulty. The primary could order requests differently for different processors, which would lead to the processors executing the requests in different orders. The approach taken by the algorithm, is to rely on the clients to detect such diversion and ignore the faulty replies. The client achieves this by contacting a *sufficient* number of processors before accepting a response.

Aublin *et al.* developed an algorithm that mimics Zyzzyva but uses significantly less amount of code when implemented [5]. The modified version of Zyzzyva was implemented in order to show proof of *Abstract* (ABortable STate mAChine replicaTion) which aimed to provide an easier way to develop BFT state machine replication algorithms. One core property of Abstract is the possibility to abort requests from clients if an error has occurred in the system. Note that neither this algorithm nor its predecessor Zyzzyva is self-stabilizing.

There exists an open-source library for implementing a BFT replicated state machine developed by Bessani *et al.* [6]. The library (BFT - SMaRt) was developed in order to provide a practical and reliable software which should support, for example, multicore-awareness and reconfiguration. The library outperforms other at the time existing libraries when it comes to number of faults tolerated.

One of the first *self-stabilizing* BFT algorithms for a replicated state machine was developed by Binun *et al.* [7]. It relies on clock synchronisation to ensure that all

processors start executing the algorithm at the same time. All processors share their information about the system, which in combination with using a Leader-Less Byzantine fault-tolerant algorithm, enables the possibility for consensus to be reached in the system.

The studied solution by Dolev *et al.* [3] did not follow the severe assumptions of [7, 8], because they imply inferior performances since their solutions require synchrony. The studied algorithm is not the only one that can provide a self-stabilizing solution for state-machine replication, e.g., self-stabilizing virtual synchrony [9] and solutions that are based on group communication [10, 11, 12]. State machine replication can also be based on consensus, thus it is worthwhile to mention the first self-stabilizing solution of such kinds [13].

1.5 Our contribution

We are the first to implement and practically validate the first self-stabilizing algorithm for a replicated state machine which uses failure detectors, while also being Byzantine fault-tolerant [3]. In order to validate the correctness of the SSPBFT algorithm, 120+ unit tests testing minor functionality and 30+ integration tests validating proof invariants, were implemented.

The SSPBFT algorithm was also deployed and evaluated as a distributed system, in order to validate the approach in a real-world-like manner. Various tooling was built to facilitate such deployment in our chosen evaluation environment. The existing codebase can easily be used for other projects, providing a foundation for building systems based on distributed algorithms. At the time of writing, two different projects at the department of Computer Science and Engineering at Chalmers University of Technology are already using big parts of the codebase.

Furthermore, we have conducted a preliminary evaluation of the implemented system. Different performance aspects have been evaluated to show the possibility of future work based on our implementation.

2
Theory

This chapter aims to explain concepts, terminology and necessary theoretical background needed in order to implement a Byzantine fault-tolerant replicated state machine. An overview of the implemented SSPBFT algorithm is presented along with an overview of high-level pseudo code.

2.1 State Machine Replication

In its simplest form, a *state machine* is a mechanism that, given a starting state S and an operation R progresses to the (next) state S' upon the execution of R. The execution of operations is deterministic, meaning that all executions of a certain operation given the same starting state will cause the state machine to progress to the same (next) state. A state machine can, for example, be used as storage as the state machine itself can be data modelled as a set of operations executed in a particular order.

State Machine Replication (SMR) provides a way to replicate a state machine in a distributed system, as described in [14]. Each replicated state machine, also called *replica*, stores its current state. In a fault-free world, all replicas will have the most recent state with all operations performed since the default state. It could also be the case that replicas have a default state but with all required pending operations to be executed to progress to the most recent state.

A visualisation of a replicated state machine can be found in Figure 2.1. The client sends a message to one of the replicas (R_2), containing the desired operation to perform on the state machine and the replica will contact the other replicas. All replicas will update their state accordingly given the operation. After completing the state update, R_2 sends a reply back to the client. This is one example of a communication pattern in SMR. Depending on how the system is implemented a client might, for example, need to contact all replicas or all replicas might send a reply back to the client.

By obtaining multiple replicas of a state machine in a distributed system, higher availability of the service can be offered to users. Users have a wider range of replicas to contact in the system, in order to perform a certain operation. By having a replicated state machine the single point of failure can also be removed which results in an even more robust system. For example, in the event of a crash failure without

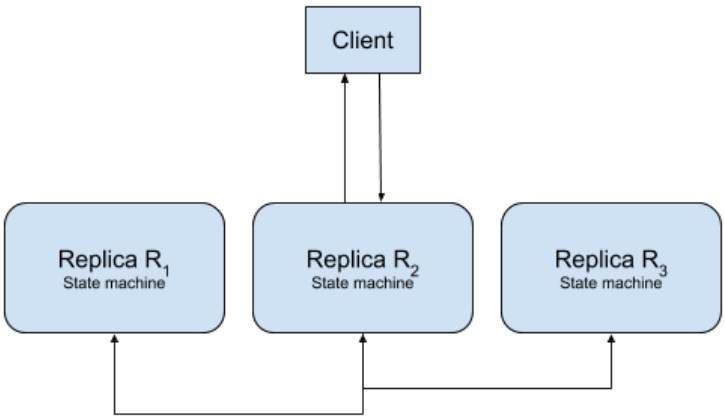

Figure 2.1: A replicated state machine containing three replicas. The communication flow starts with the client contacting replica R_2, which propagates the request to the other replicas before returning a reply to the client.

the data being replicated in the system, data might be lost. If such a failure occurs in a replicated state machine, data might still be available on correctly functioning replicas, since the data is designed to exist on all correct replicas in the system. This is one of the key benefits of using SMR.

One issue with implementing a replicated state machine is to achieve consistency among the replicas. It is highly important that the replicas contain a mutually consistent state. As mentioned before, in a fault-free system they will always contain the most recent state, but a real-world system can certainly not be considered to be fault-free. In a system containing a replicated state machine, the term *consensus* is used to describe replicas agreeing upon a state that is the current correct state [15].

2.2 Failures

A distributed system such as a replicated state machine must be able to deal with various types of failures. These failures all have varying impact on the system and can be divided into two categories with respect to the harm imposed on the system; *benign* and *severe*. Benign failures, as the name implies, does not result in as much negative impact to the system as severe failures. Apart from classifying failures with respect to how harmful they are, they can also be classified based on their duration. *Transient*, or intermittent, failures are failures that occur for a certain duration before stopping, allowing the system to continue operating without the failure being present. *Permanent* failures is the other category of failures with respect to duration and once they occur in the system they will be present throughout the rest of the

execution, unless action is taken to remove that failure.

2.2.1 Types of failures

There are many problems that may arise in a distributed system and many of them are due to failures of varying types. For example, there is always the risk of processor crashes or communication link failures. Other failures that might occur are packet failures such as omission, packet loss, duplication or re-ordering.

Apart from the mentioned failures, processors might also behave arbitrarily, or even maliciously. If so, they are said to be acting Byzantine [16]. A Byzantine fault is permanent and will remain in the system until actions are taken against it. The Byzantine processors might act according to the rules of the system for a period of time to avoid suspicion and then perform a malicious act to try to trick the system. For example, consider an attacker in the system with full control of one processor in the system, with the ultimate goal of hindering system progression. Problems occurring from the attacker being in the system will remain until actions are taken to remove the attacker, rendering the failure permanent. The Byzantine failures are also classified as severe due to the difficulty of detecting and being tolerant to them. A summary of the mentioned faults can be found in Table 2.1.

Failures	**Duration**
Processor crashes	Permanent
Communication link failures	Permanent
Packet failures; omission, duplication, reordering	Transient
Byzantine behaviour	Permanent

Table 2.1: A summary of failures that might occur in a distributed system.

In asynchronous systems, as most real-world distributed systems are, it is non-trivial to determine which failure has occurred. For example, there is no way to distinguish a packet loss from a packet delay, since in an asynchronous system it is not known how long it may take for a packet to arrive. Failures such as having a processor entering an arbitrary state of violation (Byzantine), where it is not executing correctly, increment the difficulty even more. These faults may be irregular and inconsistent, rendering them very difficult to detect.

2.3 Self-stabilization

The introduction of a self-stabilizing system requires introducing the following definitions. A *legit system behaviour*, or *legal execution*, depends on the system in question and is defined by the system requirements. Legal execution is the way a system should behave, whether it is self-stabilizing or not. However, a self-stabilizing system has the ability to reach its legal execution after the occurrence of the last *transient fault*. We assume that any transient fault leads to an arbitrary state of the

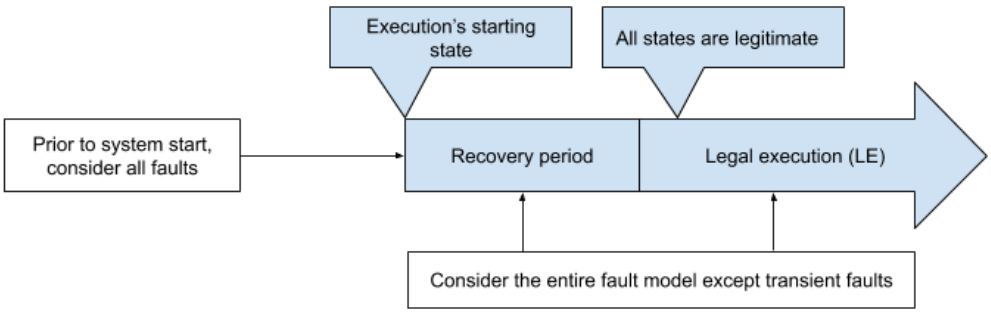

Figure 2.2: An example over a system execution with respect to when different failures may occur and the recovery guarantees.

system, possibly with stale information but with the program code intact. Failures that lie outside the fault model of the system can be considered to be transient faults as long as the program code returns to represent the algorithm. Such transient faults could lead to the system having e.g. stale variables, and can potentially affect the outcome of the system, but are not considered to affect the program code.

A system is said to be *self-stabilizing* if and only if, the system is guaranteed to reach its legal execution within a bounded number of execution steps regardless of its initial state [2]. A self-stabilizing system will automatically recover from a failure under the assumption that no transient faults occur during the system execution. A *recovery period* is considered to be the period when the system is converging to its legal execution. An example of a system execution is illustrated in Figure 2.2.

Another requirement of self-stabilizing systems is *closure*, meaning that once in a legal execution, the system will continue to exhibit legitimate behaviour in the absence of transient faults.

2.4 System model

Designing an algorithm for a Byzantine fault-tolerant system naturally requires modelling the system. The so called *Adversary model* is commonly used as a system model since it models a system with the presence of an adversary, who tries to challenge the logic of the system [17]. In a replicated state machine the adversary might force some replicas to update their state to a certain value while tricking other replicas to update their states to a different value. The system will therefore not be in consensus given that replicas are in different states and should this occur, the underlying algorithm might break. It is therefore beneficial to design algorithms using the adversary model, so that the systems may tolerate Byzantine behaviour specified by the model.

The power of the adversary is usually limited by making assumptions on, for exam-

ple, the computational capacity of the adversary. In the case of SMR, the constraint may be expressed in terms of how many of the replicas the adversary can control at the same time. In order to model this constraint, the following notation is often used:

- n : The total number of replicas in the system
- f : The total number of replicas controlled by an adversary in the system (also referred to as *faulty* replicas)

The constraint can be modelled as a threshold on the number of faulty replicas in the system and represented as a relation between the total number of replicas and the number of faulty ones. For example, a system with the constraint that the number of faulty replicas will always be less than half of the total number of replicas can therefore be defined as $n > 2f$. Bitcoin, for example, uses a similar type of constraint; if the correct replicas control the majority of the computational power, it becomes infeasible for an adversary to change the transactions[1].

2.5 Limitations

There are some proven limitations that need to be taken into consideration when designing a self-stabilizing Byzantine fault-tolerant distributed system. The following limitations are considered:

- *Fischer, Lynch and Paterson (FLP) Impossibility* [15]
- Constraint on faulty replicas [16]
- Impossibility of achieving asynchrony, BFT and dealing with stale information at the same time [18] [19]

This section aims to introduce and explain these limitations more in detail.

2.5.1 FLP Impossibility

A distributed asynchronous system cannot reach consensus, in the absence of a mechanism for determining whether or not a processor has crashed or if it simply taking a long time to respond, as proven by Fischer *et al.* [15]. This constraint, which is referred to as the *FLP Impossibility*, requires systems to either be synchronous to some extent or use randomisation, in order to reach consensus. For example, the system may have some sort of a failure detector, use clock synchronisation or other mechanisms to determine when processors are not responding. Synchrony can be either strong, meaning execution steps are taken simultaneously, or weak, where there is an upper limit on the time allowed for a step to be taken. Distributed systems with a strong synchrony come with limitations on, for example, availability, since it requires processors to wait for the slowest processors before executing the next step.

[1] https://bitcoin.org/bitcoin.pdf

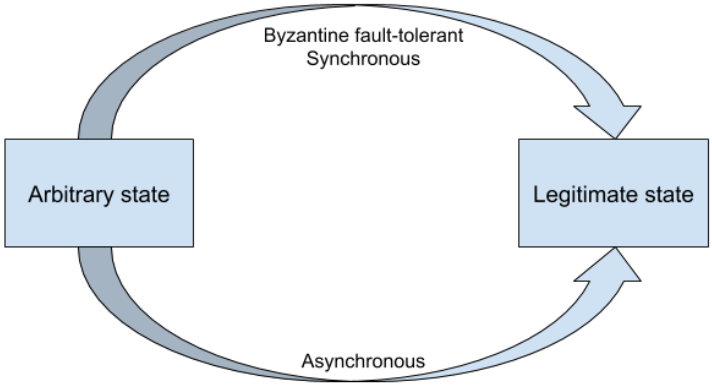

Figure 2.3: An illustration during a recovery period with different paths due to the limitation of asynchrony, Byzantine and stale information.

Strong synchrony is suitable when processors are located closed to each other where they can easily be triggered by a global clock pulse [20]. Since strong synchronous systems rarely appear in reality given the difficulty of clock synchronisation in distributed computing, systems with a weaker synchrony are usually used. Solutions to reaching consensus in such systems are found to be more complex and non-trivial to implement.

2.5.2 Constraint on number of Byzantine replicas

As proven by Lamport *et al.* [16], no more than a third of the number of total replicas in the system may act Byzantine in order for the system to be able to reach consensus. If the number of Byzantine replicas is represented as f, that means that the total number of replicas in the system n must be set to $n \geq 3f + 1$. Algorithms that function correctly under this constraint are said to be Byzantine fault-tolerant and such algorithms can act as fundamental building blocks of BFT systems.

2.5.3 Asynchrony, Byzantine and stale information

As proven by Kekkonen-Moneta *et al.* [18] and Nesterenko *et al.* [19], a system cannot be asynchronous, Byzantine fault-tolerant and deal with stale information through self-stabilization at the same time; only two out of these three properties can be fulfilled. Consequently, different recovery paths for such a system exist as can be seen in Figure 2.3. One path shows that when the system is tolerant to Byzantine, synchrony is required in order to reach the legitimate state. The other path does not require synchrony, enabling the system to be asynchronous but no Byzantine faults are assumed to occur.

2.6 Self-stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors

The following chapter provides a brief overview of the SSPBFT algorithm developed by Dolev *et al.* [3]. For further details including correctness proofs, see [3]. Note that the interpretation of *processor* here is equivalent to both *replica* and *node*. The denotation processor is used in order to provide consistency with the paper of the algorithm.

The algorithm is composed of three modules, each module responsible for a specific task. The *View Establishment module* provides a consistent agreement among the processors of which processor is the *primary* processor of the system at a certain time. Within one *view* one specific processor is assigned the primary role. It is the job of the primary to perform a consecutive assignment of sequence numbers to incoming client requests, in similarity with [1]. The logic of the assignments is implemented in the *Replication module*, which guarantees identical ordering of requests across all replicas. The third and last module, the *Primary Monitoring module*, is essentially a failure detector which checks and validates the work done by the current primary processor. If the primary processor is behaving incorrectly, the Primary Monitoring module will detect it and request a view change in the system that initiates the procedure of changing the primary processor.

In the algorithm, the constraint on the number of processors is set to $n \geq 5f + 1$, which is necessary for the view establishment procedure. However, this constraint can be reduced in order to provide an optimal resilience of $n \geq 3f + 1$, see [3]. Due to the limitations on a self-stabilizing system, as mentioned in Section 2.5.3, the algorithm needs to assume that during a view establishment no malicious processors are present. By removing the BFT constraint, view establishment can be carried out in a self-stabilizing asynchronous system even in the presence of stale information. This malicious-free assumption may be removed by introducing an event-driven failure detector presented later in Section 2.6.4, which instead imposes a weak synchrony among correct processors.

A client needs to contact all processors in the system when sending in requests. This is due to the fact that the algorithm makes information-based decisions, meaning that information is shared among the processors. The client also needs to wait for replies from $f + 1$ processors before considering its request executed, in similarity with [4].

2.6.1 View Establishment Module

The View Establishment module provides a consistent choice of primary among the processors in the system. The module introduces the notation of a *view*, in which a specific processor is assigned a coordinator role; hence the processor is referred to as the *primary*. The total number of possible views in the system is the same as the number of processors and it is predefined which processors is the primary in a

certain view. The task of the primary is to ensure total order of execution among the client requests within that specific view, which is, as mentioned, executed by the Replication Module. When transitioning from one view to another, the primary processor also changes, since only one specific processor may be the primary in a given view. A view change will either occur if the Primary Monitoring module has invoked a view change when detecting a failure in the primary's progress or when the majority of the processors has decided to progress to the next view.

In order to establish a consistent view in the system, a processor can be in two different phases; phase 0 and phase 1. In both phases the processor may wait for more processors to acknowledge the current phase and view of itself, it can conduct a reset of the module or it can move to the next phase (phase 0 to phase 1 or vice versa). In phase 0 processors are awaiting a view transition to be initiated by other modules or for conflicts in views across the system. When one of these events occurs the processor moves to phase 1, given that other processors are also aware of the event. Whilst in phase 1, the processor establishes the new view or falls back to a predefined default view. Once again, the establishment of the new view is executed only when other processors are supporting the establishment. The processor re-enters phase 0 after completing the view establishment.

The View Establishment module consists of two parts - a Coordination Automaton and a sequence of operations, suggested by the automaton. The automaton ensures that a processor does not perform a view change unless certain conditions are fulfilled. These conditions are formed in a way that at least $n - f$ processors need to agree on the view change. It is the job of the automaton to ensure that the processor is executing the right action depending on its current phase. The outline of the automaton can be found in Algorithm 1.

The second part of the module, the series of suggestions from the automaton, includes both predicates and actions. These suggestions are phase-specific and the predicates ensure that the actions are only executed if the view of the processor has been acknowledged by a sufficient amount of processors. In the case of stale information being detected, the automaton triggers a reset action to prevent stale information that could halt the progress and endanger *liveness*. Liveness means that the system always progresses.

The system needs to have a consistent view since the replicas need to agree on which processor is the primary processor (which enforces the total ordering of client requests). It is only once a primary is in place that the replication can proceed. The different phases and series of predicates and actions together enable a consistent view change, in the sense that all view changes are based upon informative decisions which the majority of correct processors have agreed upon.

Algorithm 1 High-level outline of the View Establishment Module, for processor p_i

1: **Variables:**
2: *views*: what processor p_i knows about the views of all processors (including itself)
3: *phases*: what processor p_i knows about the phases of all processors (including itself)
4: *witnessSet*: set of processors whose views and phases have been acknowledged by at least $n - f$ processors
5:
6: **while** forever **do**
7: Reset the system variables if necessary
8: Check if $n - f$ processors have acknowledged the most recent view and phase of processor p_i.
9: Update *witnessSet*
10: **if** *witnessSet* contains at least $n - f$ processors **then**
11: **if** predicate for current phase is true **then**
12: Perform the action corresponding to the predicate, processor p_i may stay in the same phase or move to the next phase
13: **for all** processors p_j **do**
14: Send current view, phase and information that processor p_i knows about processor p_j

2.6.2 Replication Module

The Replication module follows the same approach as taken in PBFT by Castro and Liskov [1], meaning that the primary processor assigns consecutive sequence numbers to the received client requests and the other processors follow this order when executing the request operations. The differences of the Replication module in comparison to Castro *et al.* lie in the effects of stale local information. In the module there are reset functions which will reset the local information at a processor if it appears to be stale. This means that a processor with stale information will eventually receive the correct up-to-date information from other processors and will therefore be able to catch up with the rest of the system. The sequence numbers being assigned by the primary are bounded by *MAXINT*, which leads to a *practically self-stabilizing* system [21]. Pseudo code for the module logic can be found as Algorithm 2.

The Replication module can only process client requests when $n - f$ processors are in phase 0 within the same view, which is accomplished by the View Establishment module. This is because once in phase 1 an establishment of a new view is in process. The views of the processors might therefore differ for a period of time before all processors have established the new view. If the processors are allowed to process client requests during a view establishment period, the processors might be following different primaries and the same request can possibly be assigned different sequence numbers across the system. This is not legit because the replicas will then

execute client requests in different order and the state machine will not be consistent.

The module utilises a 3-phase commit protocol to ensure correct execution ordering of requests across the system. Each step of the protocol requires an agreement of $n-2f$ processors in order to continue to the next step. The primary is responsible for assigning consecutive sequence numbers to the requests and broadcast each request to other processors, in a *PRE-PREP* message. Whenever other processors receive a *PRE-PREP* message from the primary and the agreement condition is fulfilled, the message is considered to be a prepared (*PREP*) message which indicates that the request of this message is ready to be committed. Processors then await agreement before shifting the request to *COMMIT* status, which is the last step of the protocol. When enough processors agree on committing, the request is executed, meaning that the operation of the client request is applied on the state machine.

2.6.3 Primary Monitoring Module

In order to stop a malicious, adversarial primary processor from preventing or jeopardising the progress of the replication scheme, the Primary Monitoring module consists of a failure detection mechanism. The job of the failure detector is to monitor the primary processor and trigger when the primary is suspected to be malicious (acting Byzantine).

The module consists of the logic of the failure detector, Algorithm 3, and the logic for when a view change needs to be executed, Algorithm 4. The demand of a view change is sent to the View Establishment module that conducts the actual view change. A correct processor may stop providing its services, meaning stop executing the 3-phase protocol, in the case where $n - 2f$ processors are demanding a view change although the processor itself may not yet suspect the primary. In this case the Replication module cannot make progress and a view change will eventually occur.

The properties liveness, as previously described, and *safety*, meaning keeping protocol rules from being violated, are important to fulfil. These properties are ensured by having the failure detector keeping track of a so called *beat*, a responsiveness pulse, and *count*, a progress pulse, for other processors. Beat is used to identify a non-responsive primary, which ensures that a primary that is non-responsive will eventually be identified and later replaced. The unresponsiveness could be due to packet delays and/or loss and not due to malicious behaviour, but the main focus of the failure detector is to suggest suspicion of a primary processor. To ensure that progress in executing client requests is made, the count variable is used to check progression in the job of the primary, i.e. that the primary is assigning sequence numbers to pending client requests. This ensures that the primary is not maliciously halting the system. This is required because a malicious primary might be sending out unrelated (dummy) messages, and therefore tricking the logic of the beat by appearing responsive, but halting the actual progress of the replication since no client requests are actually dealt with.

Algorithm 2 High-level outline of the Replication Module, for processor p_i

Variables:
pendingRequests: client requests that have not been processed
requestQueue: client requests that have started being processed
loggedRequests: client requests that have been executed
prim: id of current primary processor, corresponds to the currently known view
σ: interval for allowed sequence numbers

while forever **do**
    Check if a view change has occurred and if so, update *prim* to the new primary
    if processor p_i is the new primary **then**
        Wait for $n - f$ processors to change their *prim* to processor p_i
        Initiate 3-phase-protocol of all requests that have not yet been executed by $n - f$ processors
    else
        Check the consistency of the state and the requests processed by the new primary
    Check for state consistency, find the state based on a common prefix of executed requests among $n - 2f$ processors and on consistency in *requestQueue* and *pendingRequests*.
    if state inconsistency is detected **then**
        Reset variables and request a view change from the View Establishment module
    if no conflicts in views or replica states and processor p_i is not in a view change **then**
        if processor p_i is *prim* **then**
            Assign sequence numbers to requests in *pendingRequests* and initiate the 3-phase protocol for each of those requests. Add the requests to the *requestQueue*
        else
            Accept the assignment of sequence numbers to requests done by the primary if the numbers are within the threshold σ, there is a PRE-PREP message for the request and the content is the same for $n - 2f$ processors
        Commit accepted requests
        When a request has been committed by $n - 2f$ processors, execute the request and append it to *loggedRequests* and remove it from *pendingRequests* and *requestQueue*
for all other processors **do**
    Send state
for all clients **do**
    Send last executed request
If received request from client, add request to *pendingRequests*

Once the primary has been suspected by a processor to be corrupted, that processor keeps believing that the primary is corrupt until there has been a view change, resulting in a change of primary processor. Note that the processor still provides replication service until there is enough support for a view change. A truly malicious primary will eventually be suspected by a majority of correct processors, since all processors send their suspicions to each other. The only option to avoid being suspected is for the primary to process requests and thus make progress in the replication scheme. If it fails to do so there will be a view change, which results in the primary being replaced since each view has a different primary. Full details can be found in [3].

Algorithm 3 Outline of the Primary Monitoring Module - Failure detector, for processor p_i

Variables:
beat: responsive pulse for all processors
cnt: progress pulse for primary
T: threshold for non-responsive processor
prim: id of current primary processor

Upon receiving token from processor p_j
Let *beat* of processor p_j be reset and increment *beat* of all other processors
Update *prim* if a view change has occur
if processor p_i is not in a view change (phase 1) **then**
    if processor p_j is *prim* **then**
        Reset *count* of processor p_j if there has been a progress of client requests, if not increase the count
    Suspect *prim* as faulty if *beat* or *count* of prim is above threshold T

Algorithm 4 Outline of the Primary Monitoring Module - View change, for processor p_i

```
while forever do
    if no view change then
        if processor p_i suspects the primary then
            Demand a view change of the View Establishment module
        else
            if at least n − 2f processors demand a view change then
                Stop providing service to the primary
            if at least n − f processors demand a view change then
                Demand a view change
        for all other processors do
            Send data about view change demands from processor p_i
```

2.6.4 Extension: Event-driven Failure detector

The original assumption of all processors behaving correctly during a view establishment, can be relaxed by implementing a so called *event-driven* failure detector [20]. The failure detector assumes the following liveness property; there exists a known ratio, k, between the fastest and slowest token circulation on the data links of non-faulty processors. Due to this assumption, processors can know that when a fast processor has exchanged its token k times, a slow processor must have exchanged its token at least one time. The failure detector monitors the messages sent over the data links in order to ensure that a processor has received messages from all correct processors.

When a processor, p_i sends a message, related to the View Establishment module, the failure detector keeps track on how many tokens are exchanged between p_i and the other processors. Once k tokens have been exchanged between p_i and at least $n - 2f$ other processors, p_i can safely assume that it has received information from all correct processors. The event-driven failure detector is therefore only applicable when the malicious processors are known to have the slowest token circulation in the system. The detector provides functionality for executing actions at correct processors once they have received information from all other correct processors. The functionality of having to wait for other processors before executing actions imposes a weak synchrony to the system.

3
System

The SSPBFT algorithm has been implemented by modelling a replicated state machine using a simple list that supports append operations, called *BFTList*. BFTList runs on multiple processors, both when simulated locally and as a distributed system. The following chapter explains how the implementation was carried out and describes the different components in the system architecture.

3.1 Approach

This section describes the approach used when building the distributed system, that enabled the evaluation of the implemented algorithm. First, the development process is described which is followed by a section describing the testing methodology. All code for this project is licensed under the MIT license and can be found in our GitHub repositories[1].

3.1.1 Development process

The task of implementing a distributed system is very complex, requiring the implementation to be carried out with a high quality. To accommodate for the possibility of re-writing or switching out functionality later into the project, BFTList was built in a very modular fashion. The concept of modules was used, where the idea was to build the system in a way that makes it possible to replace certain modules, or components, with other alternatives should that be required. For example, the original communication module was replaced with another solution more suitable for the rapid development approach a few weeks into the project, and this was carried out with minimal overhead due to the modular design of the system.

Apart from working in a modular fashion, suitable programming languages and tooling was chosen to enable quick prototyping and minimal overhead. Python was chosen for the implementation, due to it being a good general-purpose programming language and previous experience. Fast prototyping and being able to validate ideas and approaches quickly were of great importance, which was an additional reason as to why Python was deemed such a good fit.

Lastly, the development process also focused heavily on testing, in order to gain as much confidence as possible when arguing about the correctness of the system.

[1] https://github.com/sspbft

3.1.2 Testing

In order to test the implementation throughout the development process, functionality for writing both *unit* and *integration* tests was implemented. Unit tests focus on testing a small piece of functionality in order to validate that the exposed functionality of the module does what it is intended to do. Integration tests focus on the system as a whole, where different components are tested together. By combining these two test approaches, bugs could be spotted more easily and sometimes quicker, which helped reducing the overall implementation time.

Since Python was used, it was decided to build the test bed upon the included testing library *unittest*[2]. It provides functionality for running unit tests with minimal configuration, reducing the overhead for setting up a unit testing architecture to almost zero. An integration test runner was also implemented on top of the existing unit test framework, which took care of launching nodes, setting up the communication between nodes, specifying starting configuration etc. This way, focus could be on writing the actual tests rather than the code used for running the tests, which was very helpful during the project. However, besides setting up the system, there also needed to be a way to emulate requests sent from clients to the system. To solve this, scripts and a simple client shell were implemented which made it possible to manually inject client request to the system. These requests were simply appends of an integer which were sent to all processors and then processed by the state machine. Software for running clients in a distributed fashion, where requests are automatically created and injected from different node, in a more real world-like fashion, was also developed which was used heavily in the evaluation of the system and will be further explained in the next chapter.

The execution of integration tests was carried out in the following fashion. First, the starting configuration for either the module or the entire system was defined. The system was then started and allowed to run on its own, with the starting configuration injected as its starting variables. The test then checked that the system reached the defined target state within a specified amount of time, which determined whether or not the test was successful. This could be generalised for all three algorithmic modules and even for the system as a whole, rendering integration tests and the confidence of correctness desirable when implementing a distributed system.

3.2 System architecture

The following section will give an overview of the system in its entirety and descriptions of BFTList including the communication modules are presented.

[2]https://docs.python.org/3.7/library/unittest.html

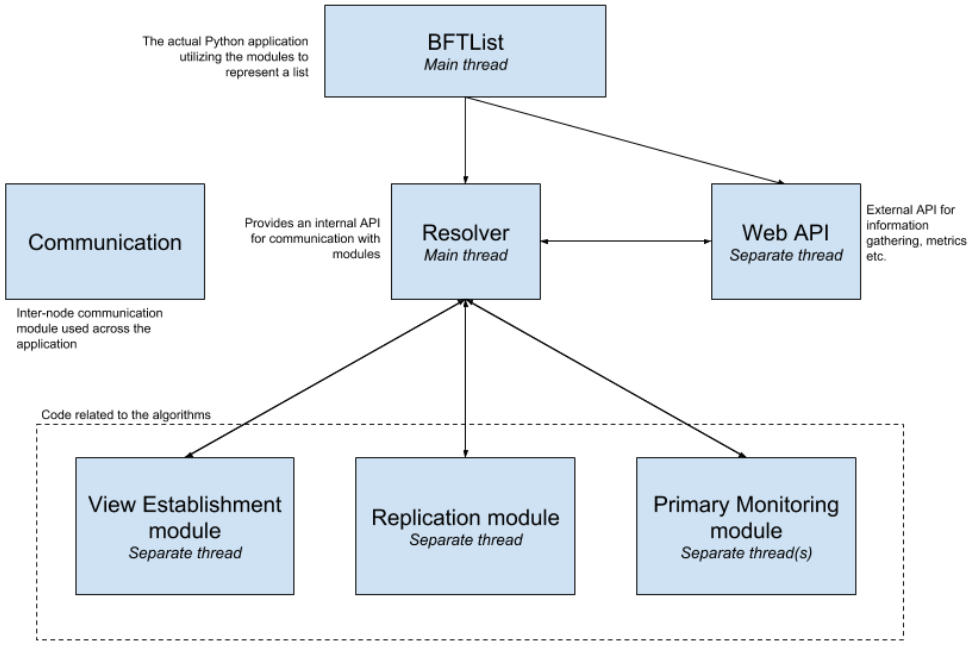

Figure 3.1: BFTList architecture. The communication module handles inter-node communication, the Resolver deals with both inter-node and inter-module communication and the Web API is used for external functionality. The code related to the SSPBFT algorithm resides in the three algorithm modules in the bottom of the figure.

3.2.1 BFTList - a replicated state machine

The replicated state machine is modelled as a list and by modelling the state machine in a simple fashion, more time could be spent on the SSPBFT algorithm and its validation. The list only supports *APPEND* operations, where each operation thus adds an element to the list, making it larger and larger as more operations are executed. This can be compared with a blockchain, where the chain of blocks continuously grows in the same fashion as BFTList.

The so called *BFTList*, representing the replicated state machine, includes several components; the main component (BFTList) including an inter-module manager component (called Resolver), three algorithm modules, a Web API and a communication module for inter-node communication. Each of these components is described in more details in the following sections. An overview of the architecture of the implementation can be found in Figure 3.1.

3.2.1.1 Resolver

Even though the implemented algorithm consists of three distinct modules, the modules should still behave as *one* unified component. All modules run in separate threads, and hence a mechanism to keep track of all threads was needed in order to provide inter-module communication. To solve this, a so called *Resolver* was implemented. The Resolver keeps track of the running threads and provides an internal API for communicating with a specific module. By having the Resolver placed between all modules and only communicating through the Resolver, the communication layer between the modules themselves could be abstracted away which reduced both complexity and code duplication. Apart from handling inter-module communication, the Resolver is also responsible for routing both incoming and outgoing messages to the correct modules.

3.2.1.2 Algorithm modules

The logical complexity and dimensions of the SSPBFT algorithm put a high demand on code architecture and structure. This issue was addressed by implementing the different modules presented in the algorithm as individual components in the system. By separating the full algorithm into modules, the complexity of both unit and integration testing could be greatly reduced and also made it possible for the modules to run independently of each other in a concurrent fashion.

The View Establishment module was implemented using one thread and while no new view establishment is required it is monitoring if the Primary Monitoring module tells it to change the current view. The Replication module is also using one thread and can be considered to be the main module in the sense that it is here the actual progression procedure of the state machine occurs. The module either keeps ordering pending client requests and executes them, or it is awaiting a new view establishment in order to continue the progress of the state machine. The Primary Monitoring module is implemented in two threads, where the first thread deals with the actual demanding of a view change once the primary is suspected, and the second thread carries out the suspicion checks. Both of these actions should be carried out concurrently, hence the need for two threads.

The event-driven failure detector, previously mentioned as an extension of the algorithm, was implemented as one module running in a separate thread. The module only interacts with the View Establishment module which is done through the Resolver.

Names of methods and variables, data structures and code structure closely follow the structure given by Dolev *et al.* in order to be consistent the pseudo code in the paper [3]. Deviation was of course bound to happen because of edge cases where the logic of the hand-written pseudo code was not applicable to a real-world code implementation. In cases like this, the new logic was discussed with the authors of the paper to make sure that the logic did not violate the conceptual logic of the algorithm.

3.2.1.3 Web API

A web API was implemented and exposed for external services for purposes such as information gathering, remote procedure calls and so on. In order to implement such an API quickly, the micro-framework Flask was used[3]. Flask provides functionality for setting up a Web API with minimal effort, which made it a good fit for this project. The API is used by various other services related to testing, deployment and more. By providing an API in this manner, the coupling between different services could be reduced and it is easier to add additional services later on should that be needed. Such additional services do not need to be tied into the original system and can function solely based on the web API, hence making extensions feasible.

3.2.2 The *NO_OP* automaton

Since BFTList only executes *APPEND* operations, a slightly different type of state machine was also implemented; the *NO_OP* automaton. The only difference between the two implementations is the type of operation they support. The *NO_OP* automaton supports a special *no operation (NO_OP)*, meaning that the list is not modified at all. This automaton was used during the preliminary evaluation phase in order to measure the performance of the algorithm without letting the state machine increase in size with each request, which slows the system down. The *NO_OP* request type was therefore used in order to not let the choice of state machine affect the performance.

3.2.3 Communication module

Processors in the system need to be able to communicate with each other. Efficient monitoring failure detectors, can in an elegant way be implemented over UDP/IP. Message exchanges regarding monitoring liveness are therefore carried out in a self-stabilizing manner based on UDP. Other message exchanges are sent over a TCP link in order to achieve reliable communication without self-stabilizing properties. This section first gives an overview of the general communication architecture and then the two communication protocols used in the system are described.

3.2.3.1 Architecture

Each processor sets up $2(n-1)$ sender channels, meaning $n-1$ channels for each communication protocol. Each sender connects to the appropriate receiver running on the processor that the sender is intended to send messages to. A processor also establishes 2 receiver channels, one for each communication protocol, which the senders on the other processors can connect to. The communication network can be visualised as a fully connected graph, which can be seen in Figure 3.2. The receivers on each processor are responsible for receiving messages from the senders and routing them to the Resolver, which in turn routes them to the correct module of the incoming messages. Upon start, the system waits until all communication channels

[3] http://flask.pocoo.org/

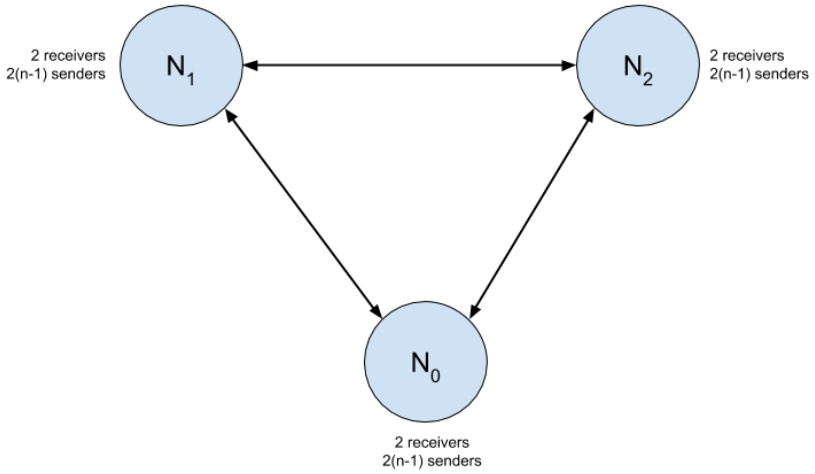

Figure 3.2: Visualisation of the communication in a network with three nodes.

have been established before starting to send messages. By starting the communication channels in a synchronised fashion, no timeouts in the communication will be triggered nor will message queues build up with messages waiting to be delivered to the processors that have not yet started and the system can function properly faster. Both receiver channels and the sender channels over UDP are running in separate threads, whereas the sender channels over TCP are constructed in an event-driven fashion using the built-in asynchronous I/O library *Asyncio*[4].

Once the system is running, each algorithm module throttles based on the number of messages waiting to be sent to other nodes in order to keep the message queues from growing in an uncontrolled fashion. This is done by having each module check the number of its messages still waiting to be sent and if this number is above a certain threshold, the module sleeps until more messages have been sent and the number is below said threshold.

3.2.3.2 ØMQ over TCP

ØMQ, pronounced *zero-em-queue*, provides high-speed distributed messaging with easy-to-use APIs, that remove much of the complexity often encountered when working with network programming[5]. ØMQ recommends using TCP as transport protocol and when doing so, a sender/receiver communication protocol can be implemented with ease. This helped speed up the implementation of the communication channel based on TCP greatly. Since this protocol was decided to be the primary communication solution, being able to implement it in a reliable fashion quickly was

[4]https://docs.python.org/3/library/asyncio.html
[5]http://zeromq.org/

of great value. ØMQ also comes with a ready-to-use, event-driven implementation of TCP communication links, which was introducd to the implementation in an effort to reduce the number of threads.

Each sender is responsible for sending messages to a receiver running on another processor in the network. The sender does not know anything about what type of messages should be delivered to the receiver, it only focuses on delivering the messages as quickly as possible. A sender operates based on a message queue attached to the sender in which messages can be added through the Resolver. Whenever there is a message in the queue, the sender removes that message from the queue and blocks until the message has been successfully delivered and acknowledged by the receiver through help of ØMQ's APIs. This protocol is used by all algorithm modules for communication aside for the Failure Detector, which instead uses the UDP protocol.

We note the existence of self-stabilization TCP/IP-like protocol [22] as well as a self-stabilizing Byzantine fault-tolerant end-to-end algorithm [23]. However, their implementation is outside of the scope of this project.

3.2.3.3 Self-stabilizing UDP

The self-stabilizing communication channel over UDP is based upon a token passing algorithm by Dolev [20] (p. 76). The token-passing algorithm consists of two separate algorithms, one for the sender and one for the receiver, which can both be found in Appendix B. This communication protocol is used by the Failure Detector module running on each processor, which monitors the other processors to see if they are responding to messages.

The token-passing algorithm makes use of a counter, referred to as a token, which is sent back and forth between the sender and receiver using UDP as transport protocol. Whenever the sender sends a message, it waits until it gets back the last sent token before sending the next message. Should the token not be returned within a specified amount of time, a timeout is triggered and the last sent message is re-sent with the same token. Whenever the token is returned to the sender, it is incremented by one before sent back once again, which ensures that no two messages can be sent with the same token and hence no re-transmission of already sent and delivered messages is possible.

4
Evaluation

The first key part of the project was to validate the correctness of the SSPBFT algorithm. This was achieved by emulating different failures and scenarios along with validating the functionality while doing so. The second key part consisted of performing a preliminary evaluation, with respect to the performance of the implementation. This chapter presents the different evaluation approaches as well as how they were analysed.

It should be noted that due to the asynchronous nature of the studied distributed system, we do not expect our validation efforts to be complete in the sense that they could fully substitute the written proof or a verification method using formal automated analysis. These efforts serve as a practical validation of the proof invariants and are intended to complement the written proof.

4.1 Failures

Validating the implemented SSPBFT algorithm involved proving tolerance against failures, such as the ones described in Section 2.2. Some failures, which the algorithm claims to be tolerant against, have not directly been emulated. For example, the algorithm has not been validated with respect to packet failures or fault injection on the underlying transport layers; the evaluation of these already existing system components was not within the scope of this project. Nevertheless, it was deemed fair to assume that such failures occurred in the system once deployed to a number of physical servers, due to their distributed nature. Listed in Table 4.1 are failures which have been emulated as a single failure. The failures are specific for the studied algorithm and are divided into different categories.

Invariants of proofs

The validation of the algorithm was performed based on invariants found in the written proofs of the algorithm [3]. These invariants can be divided into two subgroups; those that were tested on one processor and those who required the coordination of multiple processors. The first subgroup can be seen as invariants which could be tested through unit testing of the modules, e.g. when a processor by itself should realise that its current state is wrong and then correct itself, possibly by removing faulty values. The second subgroup contains invariants where communication between processors was needed in order to recover from the current failure. For

Failure	Scenario	Emulation method
Processor crashes	Permanent crash	A processor stopped responding, through code
Byzantine behaviours	Primary was non-responsive to all	The primary processor stopped responding
	Primary was non-responsive to some	The primary stopped responding to a certain amount of processors and was responding accurately to the rest
	Primary assigned faulty sequence numbers	The Byzantine primary processor sent non-consecutive sequence numbers or different sequence numbers to different processors
Corruption of variables	Inconsistent starting views	Specified different views for different processors
	Inconsistent starting states	Specified different replica states for different processors
	Inconsistent pending requests	Specified different pending client requests for different processors
	Different requests used to check progress done by primary	Specified different sets of requests that the primary assigned sequence numbers to, for different processors

Table 4.1: An list of some of the emulated failures.

example, failures such as a primary acting Byzantine and consequently trying to trick the system falls under the second subgroup since the correct nodes need to communicate in order to detect the Byzantine behaviour.

The invariants that required distributed coordination were validated by defining a starting system configuration, that was injected into the processors upon start. When the configuration was loaded for all processors, the system started executing and it was then validated that the system reached a desired state within a configured amount of time. The system was also evaluated with respect to self-stabilization, which involved validating that the core property of a self-stabilizing system was fulfilled; the system recovers to a safe configuration state no matter the starting configuration.

The validated invariants, with a description including the performed testing and validation, are listed in Appendix C. Note that some of the invariants have been implemented as several integration tests in order to test different cases. For example, tests where the primary was acting Byzantine when assigning sequence numbers were implemented and tested both with only the Replication module (validated by non-progression in the state machine) and with all modules running concurrently

(validated by the occurrence of a view change).

4.2 Global view

In order to easily get an overview of the global state in the system during test runs, a *Global view* was implemented. This unified view was essentially a visualisation of relevant variables, depending on what functionality was being tested. Using the Global view, interesting data could be inspected with ease in a coherent way without having to access and switch between the logs of all running processors. This enabled quick debugging and provided information necessary to follow the progression of the system during the test runs.

Data from the algorithm modules was extracted and used to generate HTML which along with CSS resulted in the web page that was the Global view. This was then served by the Web API on each processor, which enabled anyone with access to a processor in the system to quickly get an overview of the entire system. The data was then continuously fetched from the API again from the browser using JavaScript, enabling live update of information.

The automatic live update of system data and information enabled a quick overlook and validation that the system was progressing in the desired way. It should be noted that the Global view was not used for actual correctness validation, but merely used as a help to gain a deeper understanding of the current system state during debugging. A screen shot of the Global view for the View Establishment module from one of the test runs can be found in Figure 4.1.

4.3 Different environments

The evaluation and testing were mainly performed in two different environments; a local environment run on a single computer and a real-word environment called PlanetLab EU[1]. This section provides an introduction to these environments and a description of how the evaluation was carried out along with an overview of the evaluation architecture that can be found in Figure 4.2.

4.3.1 Local

A local environment was set up during the start of the project, to facilitate for fast prototyping and development. By not having to deploy the system to external servers prior to testing a new feature, the overhead of implementation could be reduced and helped speed up the implementation. The local environment simulated multiple servers running on the same network, by starting local processes on a

[1] https://www.planet-lab.org

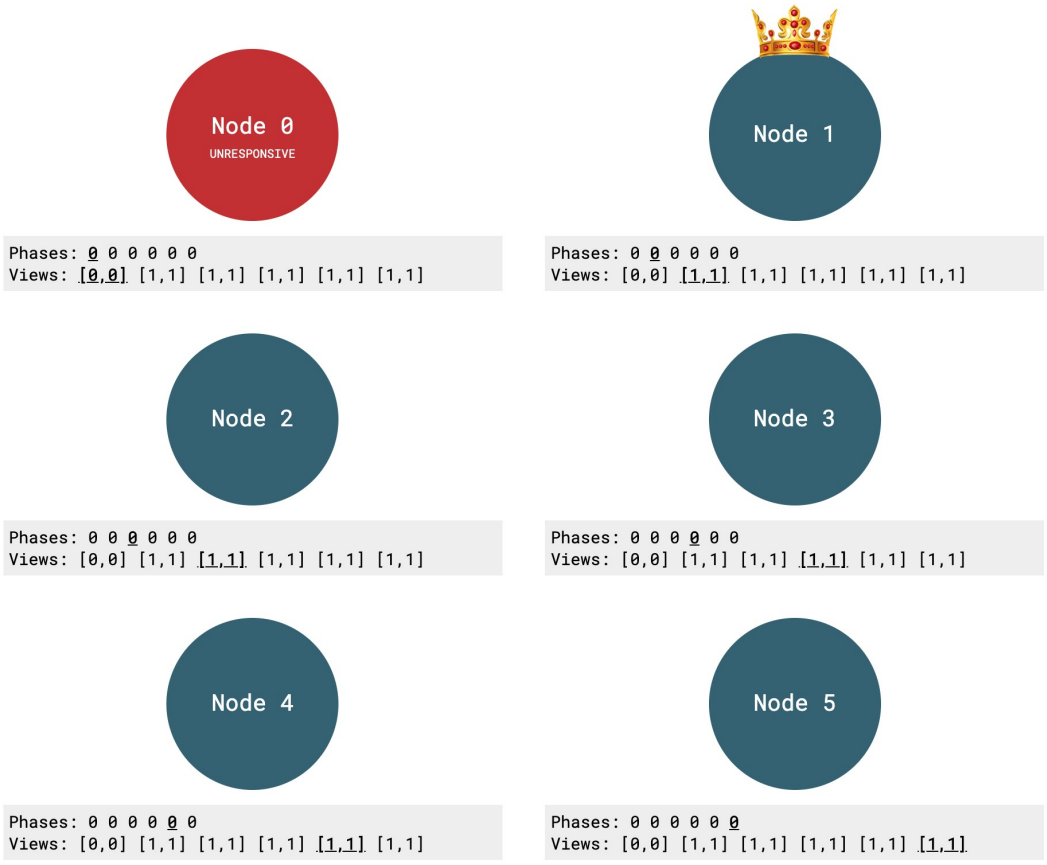

Figure 4.1: A screen shot of the Global view for the View Establishment module. Node 0 is Byzantine and exhibiting a Byzantine behaviour called "UNRESPONSIVE" and processor 1 is the primary, hence the crown. The corresponding view for each processor is shown in the grey box below each processor.

single computer, which all communicated with each other. This resulted in the system performing very well and reliable due to the minimal latency and overhead in message exchange, which was helpful to perform basic validation of the system quickly.

4.3.2 PlanetLab

To simulate a more realistic environment when testing system performance, the PlanetLab EU platform was used. PlanetLab provides hundreds of servers, also referred to as nodes, where applications such as BFTList can be deployed and tested. The deployment was done in order to ensure that the application was resilient to problems that arose when deploying in a distributed system, such as packet failures and latency. In order to make deployments to PlanetLab more efficient and easier to work with, a tool was developed that integrated with the PlanetLab API[2]. The

[2]https://www.planet-lab.org/doc/plc_api

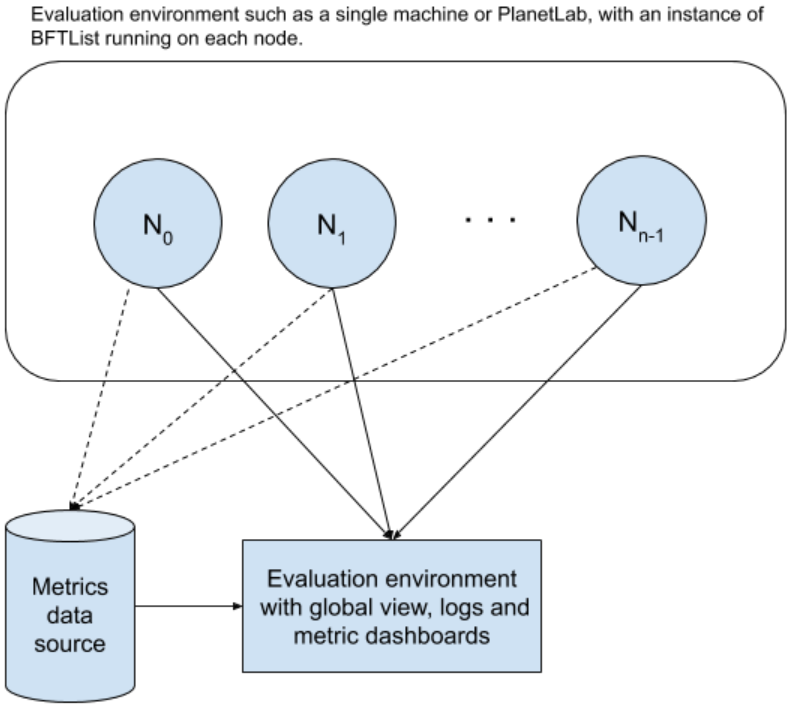

Figure 4.2: Overview of the architecture of an evaluation environment.

tool provided features such as finding nodes available for deployment, running various checks on nodes to see if they were considered to be functioning correctly and automatic deployment of applications such as BFTList. Apart from these features, various options were also available such as scaling up to a certain amount of nodes.

Apart from finding out which nodes to use in the deployment and actually performing the deployment, the logs from each node needed to be aggregated to be able to extract insights and information from the system. Without some kind of centralised logging, it would be necessary to connect to each server and check the logs manually which would not be feasible once scaling the system to more than a few nodes. Papertrail[3] provides a cheap and easy-to-use solution for this exact use case and was integrated in the deployment. This enabled searching the logs in a centralised location with a query language, using filters and other tooling to find out what was happening at the nodes.

The steps taken to perform a deployment on PlanetLab are summarised below.

1. Configure deployment settings

[3]https://papertrailapp.com/

2. Node discovery and health check using PlanetLab API
3. Deploy and start BFTList on selected nodes
4. Logs available on Papertrail and global view on each node

4.4 Metrics

When deploying an application to a set of servers in a distributed fashion, metrics can be used in order to gather information about the system health, certain events and other insights that might be important to get a full picture of how the system is performing. Since the application was deployed on an arbitrary amount of nodes, examining these things manually was not feasible and hence metrics were gathered and examined using a visualisation tool. Prometheus[4] was selected as the metric gathering software due to the fact that a Python client was available, which enabled the implementation to be completed quickly. There are different types of metrics that can be used when monitoring a system using Prometheus[5]. The two most basic ones are *Counter*, which tracks a monotonically increasing value, and *Gauge*, which tracks a value that can both increase and decrease. For example, a Counter metric can be used to count the number of executed requests in the system, whereas a Gauge can be used to measure the time taken to execute each request.

To facilitate inspection and to gain insights from the gathered metrics more quickly, a visualisation tool was needed that could create graphs and other visualisations of data. Grafana[6] was chosen as the visualisation tool, since it provides software for creating dashboards with panels that render graphs based on metrics from data sources such as Prometheus.

4.5 Experiment settings

In this section the validation of the SSPBFT algorithm and the preliminary evaluation are presented. The purpose of each different preliminary evaluation aspect is explained. First, the basic setting of all experiments is presented and then the experiments, for the specific aspects, are described. Also presented is how the experiments are conducted in order to enable such a preliminary evaluation.

4.5.1 Basic settings

Validating the algorithm was conducted on one single machine through using virtual scaling by launching separate processes emulating multiple instances of BFTList, as described in Section 4.3.1. All performance experiments were performed with either BFTList or the *NO_OP* automaton deployed to either six, nine or twelve physical

[4]https://prometheus.io/
[5]https://prometheus.io/docs/concepts/metric_types/
[6]https://grafana.com/

PlanetLab nodes. A description of the physical PlanetLab nodes used in the experiments can be found in Appendix D.

The lower bound of six physical nodes was due to the constraint on number of nodes in the algorithm, where $n \geq 5f + 1$ and the minimum value of faulty nodes, $f = 1$ was chosen. It should be noted that during some experiments no Byzantine behaviour was emulated, meaning all nodes were acting correctly.

The limitation of twelve physical nodes was due to severe problems in finding suitable servers on PlanetLab. The status of the servers varied greatly - some servers could not be accessed, some were shipped with software that was more than ten years old and some did not allow for the opening of ports etc. The demands put on the servers for running BFTList resulted in only a handful of nodes being suitable for usage, even though PlanetLab offers hundreds of machines.

Some experiments required clients to be emulated, in order to send client requests to the system. Clients were emulated in a distributed manner, where each physical node simulated one client and all simulated clients send the same number of requests. The measured execution time for a client request was considered to be the total time taken from that the request has arrived at a node until it is applied. This was measured at each node, for each injected client request. The slowest of the $f+1$ fastest execution times for each client request was considered to be the execution time for that specific request. This was due to the fact that the implemented algorithm assumes that a client waits for $f + 1$ responses before considering its request to be executed.

4.5.2 Validation of the correctness of the algorithm

The SSPBFT algorithm correctness tests were constructed to validate the functionality of the algorithm. A practical validation was necessary since the theoretical proof of is algorithm are not sufficient to show practical usage. Performing validation of the implementation was also essential in order to confirm the correctness of the algorithm.

The invariants found in Appendix C were implemented and tested in the local environment, both as unit and integration tests. All invariants were validated with six, $(n = 5f + 1)$, processors running concurrently on the same machine, except the second case of Lemma 20 which uses twelve processors rather than six.

4.5.3 Performance experiments: executing client requests

Three different approaches were taken in order to perform a preliminary evaluation regarding executing client requests:

- Executing client requests with unbounded BFTList and bounded request queues

- Executing client requests with unbounded BFTList using unbounded request queues
- Executing client requests using the *NO_OP* automaton

In a self-stabilizing system no parameter is allowed to be unbounded, hence the last part was conducted in order to see how much such property would affect the system.

All experiments in this part of the evaluation were run on six, nine and twelve physical nodes five times and during each experiment 204 client requests were injected. The exact number 204 was chosen since the number of requests needed to be divisible by six (six clients running distributed, all injecting requests) and hence 204 became the chosen number. For each client request the fastest and the slowest execution times were removed to minimise the effect of outliers. The average execution time from the remaining three test runs, for each client request, was calculated in order to analyse the performance. Standard deviation was also calculated, to get some insights as to how "noisy" the results were.

Experiments with respect to executing client requests were also conducted with two implementations of BFTList - the original self-stabilizing implementation and a non-self-stabilizing implementation. In a similar way, two versions of *NO_OP* automaton were also implemented. This was done in order to conduct a preliminary evaluation of the overhead introduced by the self-stabilizing property. The non-self-stabilizing version does not contain the View Establishment module nor any type of stale information detection or removal in any module. This version was launched with a predefined starting state, where all processors were in a stable view with a configured primary to avoid an arbitrary starting state. Due to the fact that the View Establishment module was removed, the predefined view is a requirement for correct functionality of the non-self-stabilizing version.

4.5.4 Performance experiments: code profiling

It is important to understand which functions and which parts of the system have the longest execution time. Therefore, a so called *code profiling* was performed. Profiling includes performing an analysis of the different functions in the system by, for example, counting the number of calls to a specific function or measuring the time spent in the analysed function. Such analysis of the code can be used to identify bottlenecks and parts which can potentially be optimised. Yappi[7] is a profiling tool for Python which supports multi-thread analysis and was chosen to be used during this part of the evaluation. Adding profiling tool such as Yappi to the codebase comes with a signifacnt overhead; the system will run significantly slower. Due to this overhead the experiment was conducted locally with six virtual nodes and only twelve client requests were injected.

Additionally, the run time and iterations of the do-forever loops in the algorithmic

[7]https://pypi.org/project/yappi/

modules were also measured. The three main algorithm modules (the View Establishment module, the Replication module and the Primary Monitoring module) all have a do-forever loop. The send function, where the messages are being prepared, and the throttling time in each of the three modules were also measured to get a deeper understanding of where the modules spend most of their time. This experiment was conducted with BFTList and with six virtual nodes running locally. 204 client requests containing *APPEND* were injected and three runs of the experiment were conducted.

4.5.5 Performance experiments: convergence time

These experiments measured the convergence time of the system, an important property of a self-stabilizing system. These experiments were only conducted with BFTList, since no client requests are executed. For BFTList, convergence time is considered to be the time taken for a view establishment procedure. At node level, the convergence time was measured as the time from when a view change was suggested by the Primary Monitoring module until the node had established a new view. The resulting convergence time was considered to be the slowest convergence time of the $n - f$ fastest. The experiments were conducted by having the system starting in a safe state, then emulating a Byzantine behaviour of unresponsiveness at the primary, which was eventually detected by the failure detector in the Primay Monitroing module, which enabled the measurement of convergence time to begin.

The measurement of convergence time was conducted with two different settings, the system with and without the event-driven failure detector. The experiments were performed on the same six, nine and twelve physical nodes on PlanetLab, to facilitate a fair comparison of the two settings. Each experiment was run five times and the average of the convergence times during these runs, with the slowest and fastest time removed, was considered to be the final result.

5
Evaluation results

The following chapter presents the validation and the preliminary evaluation of the SSPBFT algorithm. Firstly, the validation of the correctness is presented, showing the practical correctness of the algorithm. Thereafter the results from the client requests execution experiments are presented in the following order; experiments with unbounded BFTList using bounded queues, experiments with unbounded BFTList using unbounded queues and thereafter experiments with the *NO_OP* automaton. Then the results from the performance experiment with code profiling are presented and lastly the results from the performance experiment regarding convergence time can be found.

Our results show that the SSPBFT algorithm is functioning correctly with respect to all implemented tests based on the proof written by Dolev *et al.* [3]. The performance evaluation shows that the execution time for requests when using the main implementation BFTList grows with respect to the number of nodes in the system, but also as more and more requests are executed. However, when evaluating the *NO_OP* automaton, the execution time is shown to only grow with respect to the number of nodes in the system. This demonstrates the impact the type of state machine may have on the overall system performance. When it comes to convergence time, establishing a stable view can be done in a few seconds for optimal system settings and shows promising results for production usage. Furthermore, the results also show that most time is spent on work related to inter-node communication. This, up to some extent, was expected due to the rapid development and prototyping during this project.

5.1 Validating the correctness of the algorithm

The paper by Dolev *et al.* presenting the SSPBFT algorithm offers an extensive, theoretical correctness proof [3]. Based on invariants found in the proof, tailored tests and experiments have been constructed. These are used to further validate the SSPBFT algorithm, but in a more experimental fashion. This is performed by implementing and running a number of tests and a complete list of the invariants used can be found in Appendix C. The results show that the algorithm is behaving correctly with respect to the different implemented tests. As mentioned before, complex edge cases that had not been considered and smaller bugs that came up during the implementation were discussed and addressed together with the authors of the paper. This included both contributing to the paper and constructing additional tests.

Some variants were tested locally, i.e. no distributed coordination was needed to validate the invariant, and some in a distributed manner. An example of an invariant that was tested in a distributed manner is the invariant based on Theorem 11 in [3]. It considers closure, i.e. once the system is in a stable view, it will remain in a stable (possibly new) view. In order to test this, the system was configured such that all processors already had agreed on the same view. The system is then started and left to run for a configured amount of time, before the test finishes and validates that all processors agree on the same view. Since the system might agree on a new view, coordination between processors during the test was necessary in order to do so.

An invariant that was tested locally is based on Claim 12 in [3], which considers stale information removal through a reset mechanism in the Replication module. The test was structured such that stale information, for example an out-of-bounds sequence number, was injected to the Replication module at a processor and then the processor was left to execute the algorithm. It was then validated that the module performed a reset due to the stale information found at the processor. This did not require distributed coordination between multiple processors and could therefore be tested locally.

It should also be noted that correctness of the replication, with respect to correct execution order of client requests, was also further validated through other evaluation experiments with injection of a larger number of requests.

5.2 Performance: executing client requests with unbounded BFTList and bounded request queues

Experiments were conducted with the main BFTList implementation, meaning that client requests contained the *APPEND* operation. This was done in order to examine how a state machine continuously growing in size would affect the performance. Two versions of BFTList, one self-stabilizing and one non-self-stabilizing, were used during these experiments in order to get an indicator of the overhead of self-stabilization. A continuously growing list results in more time spent finding stale information as well as an increasing size of the messages sent. Therefore, it is expected that both versions will have longer and longer execution times as requests are being executed. The results from the experiments can be found in Figure 5.1, Figure 5.2 and Figure 5.3 with corresponding standard deviations found in Table 5.1. The trendlines corresponding to each of the versions, self-stabilizing and non-self-stabilizing, are the linear equations minimising the sum of the squared distances from each data point to the trendline.

The gap between the trendlines for the self-stabilizing and non-self-stabilizing versions grows larger and larger as the number of executed requests increases. For six

nodes there is almost no difference between the two versions at the beginning and the gap grows to around two seconds at the end of the experiment. The graph for nine nodes, in Figure 5.2, has a similar appearance. The two trendlines have an increasing gap between them starting at around 0 seconds, growing to around four to five seconds as requests were executed. For twelve nodes the gap grows at a faster rate as the requests were being executed. Both versions have a trendline that starts right above zero seconds and grows as request are executed. The trendline for the self-stabilizing version grows up to almost 100 seconds while the non-self-stabilizing version grows up to just below 50 seconds. Since the *APPEND* operation was used during this experiment the list kept growing as more and more requests were executed. The growing execution times for both versions are therefore an expected result, since all messages transferred in the system related to the system state were continuously growing as well. The trendline of the self-stabilizing version has a steeper slope, which is probably due to the extra procedure of checking stale information. The execution times are, as expected, longer for experiments with more nodes since a higher number of nodes need to cooperate in order to execute the requests.

Number of nodes	**Version**	**Standard deviation**
6	Self-stabilizing	7,50 %
6	Non-self-stabilizing	8,35 %
9	Self-stabilizing	24,94%
9	Non-self-stabilizing	20,13%
12	Self-stabilizing	10,90%
12	Non-self-stabilizing	9,47%

Table 5.1: The average standard deviation for each requests execution time during experiments with BFTList.

The message and data distribution for the experiments with the self-stabilizing version can be found in Table 5.2. It can be seen that the View Establishment module and the Primary Monitoring module have the highest message rates, since they combined are responsible for around 70-80% of all messages sent in the system. Looking at the amount of data sent by each module, one can clearly see that the Replication module is responsible for over 90% of the transferred data during experiments. This is due to the growing size of the state machine, since the whole list together with other information related to the state is sent in each replication message. The corresponding distributions for the non-self-stabilizing version can be found in Table 5.3. The Primary Monitoring module still sends the most messages and the Replication module is responsible for sending the largest amount of data, sending over 96 % of the total data due the same reason as for the self-stabilizing version. Another way of visualising the message and data distributions can be found in Figure 5.4 and Figure 5.5.

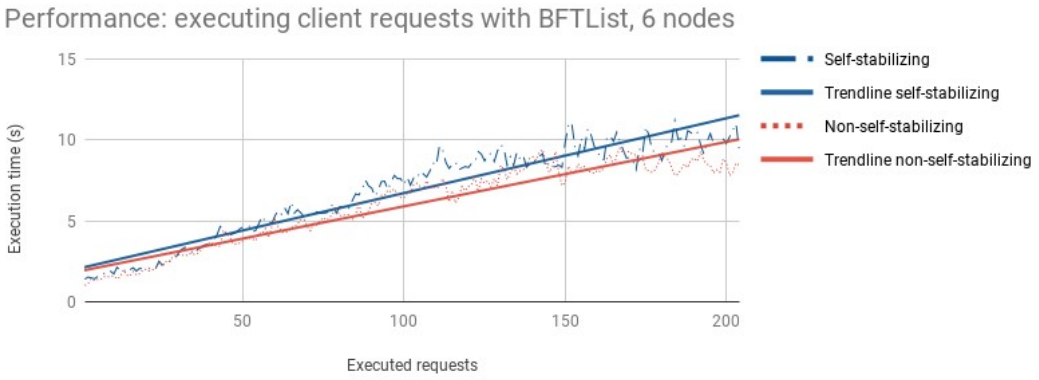

Figure 5.1: Client requests execution using BFTList for six physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,0461x + 2,11$ and the lower one for non-self-stabilization with equation $y = 0,0397x+1,93$. The trendlines grow from around two seconds to around ten seconds.

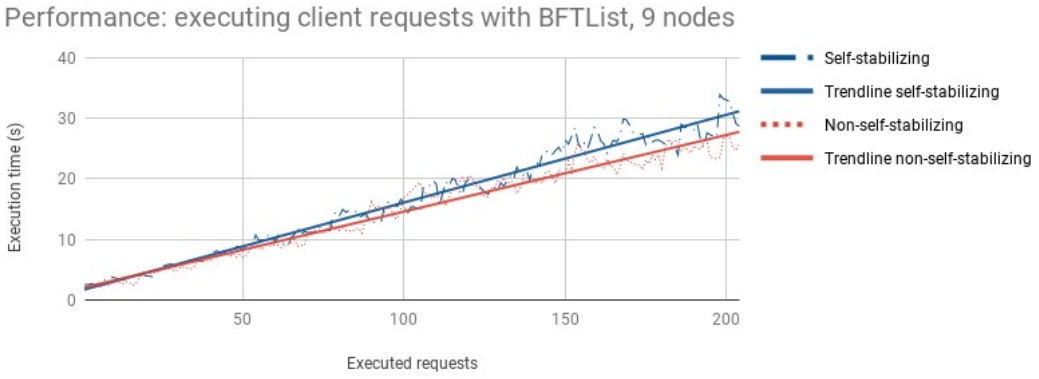

Figure 5.2: Client requests execution using BFTList for nine physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,145x + 1,65$ and the lower one for non-self-stabilization with equation $y = 0,126x+1,99$. The trendlines grow from around 2 seconds to around 28-31 seconds.

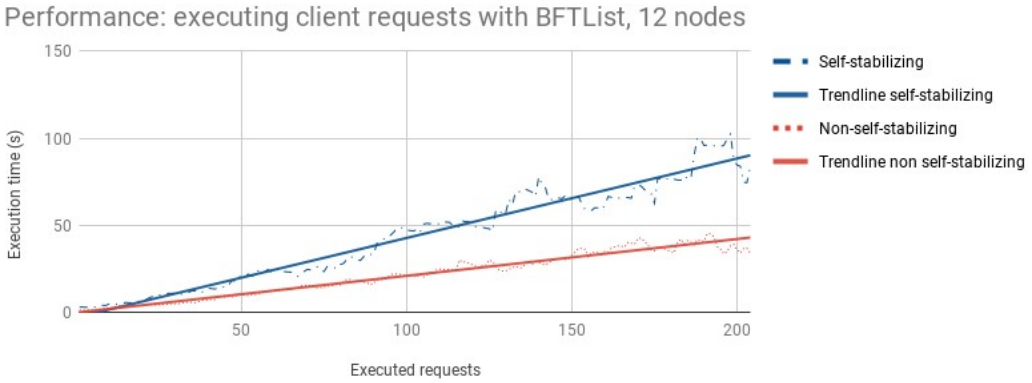

Figure 5.3: Client requests execution using BFTList for twelve physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,456x - 2,96$ and the lower one for non-self-stabilization with equation $y = 0,211x - 0,176$. The trendline for the self-stabilizing version grows from just above zero seconds to around 90 seconds. The non-self-stabilizing version has a trendline that starts just above zero as well, but only grows to around 45 seconds.

Message distribution				
n	**VE**	**R**	**PM**	**FD**
6	31,4%	19,7%	38,3%	10,6%
9	32,8%	20,2%	38,3%	8,7%
12	37,2%	13,7%	42,3%	6,8%

Data distribution				
n	**VE**	**R**	**PM**	**FD**
6	2,2%	94,6%	2,9%	0,3%
9	2,0%	95,3%	2,5%	0,2%
12	2,8%	93,6%	3,4%	0,2%

Table 5.2: The message type and data distribution during experiments run with the self-stabilizing version. Notation: Number of nodes (n), View Establishment module (VE), Replication module (R), Primary Monitoring module (PM) and Failure Detector (FD).

Message distribution				
n	**VE**	**R**	**PM**	**FD**
6	-	27,5%	56,6%	15,9%
9	-	30,1%	56,6%	13,3%
12	-	28,6%	57,9%	13,5%

Data distribution				
n	**VE**	**R**	**PM**	**FD**
6	-	96,9%	2,8%	0,3%
9	-	97,5%	2,3%	0,2%
12	-	97,7%	2,1%	0,2%

Table 5.3: The message type and data distribution during experiments run with the non-self-stabilizing version. Notation: Number of nodes (n), View Establishment module (VE), Replication module (R), Primary Monitoring module (PM) and Failure Detector (FD). Note that the View Establishment module is not part of the non-self-stabilizing version, hence the absence of results for this module.

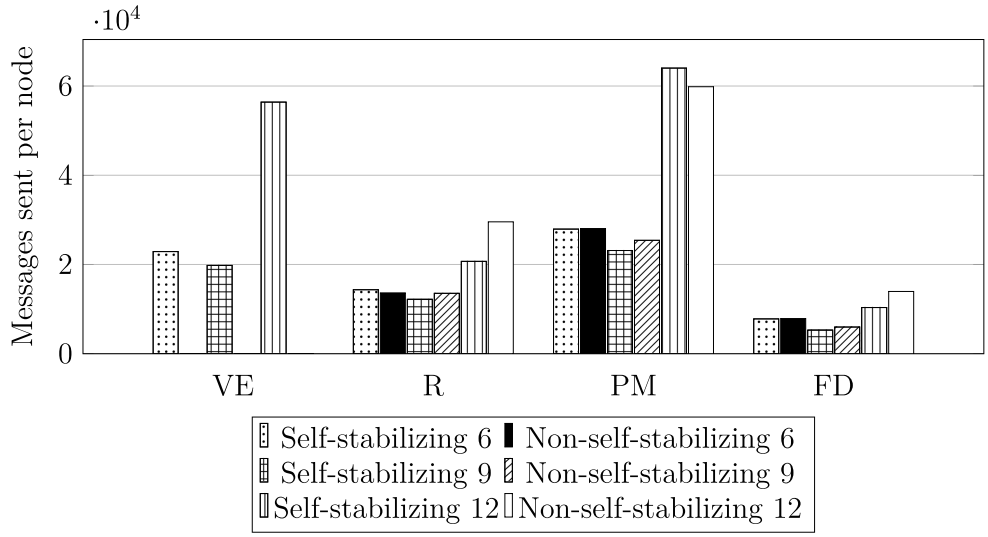

Figure 5.4: Message type distribution. Notation: View Establishment module (VE), Replication module (R), Primary Monitoring module (PM) and Failure Detector (FD).

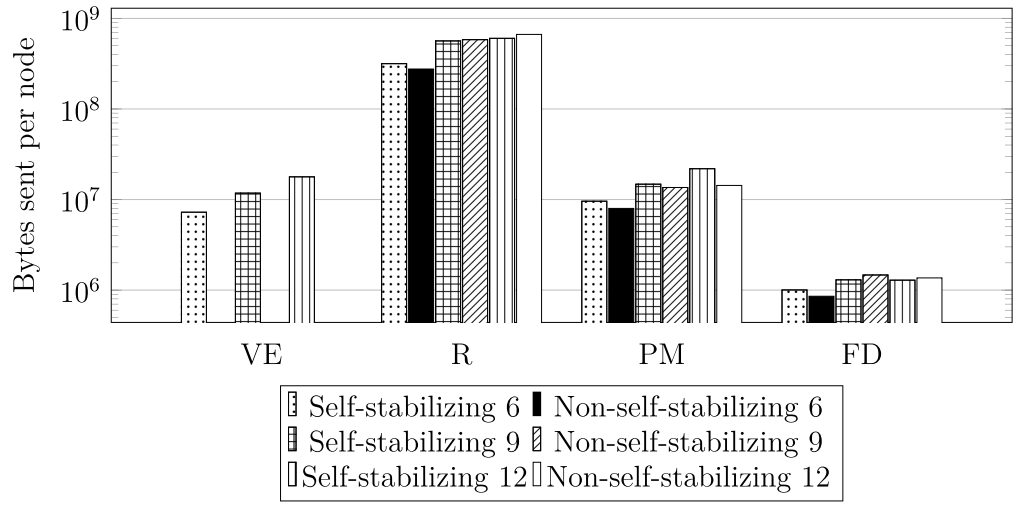

Figure 5.5: Data distribution with respect to the different messages types. Notation: View Establishment module (VE), Replication module (R), Primary Monitoring module (PM) and Failure Detector (FD). Note the logarithmic scale on the y-axis.

5.3 Performance: executing client requests with unbounded BFTList and unbounded request queues

Experiments were also conducted without bounding the request queues in the Replication module in BFTList. This was done in order to examine how much such a

bound would affect the performance of client requests execution. One would expect that the unbounded version has a faster increase in the execution time growth than the bounded version due to more data being transferred in the system. The results can be found in Figure 5.6, Figure 5.7 and Figure 5.8.

Similar to the previous experiments with BFTList, the list kept growing as more and more requests were executed, but the request queues were now also increasing in size. The gap between the trendlines for the two implementations is also increasing as more and more requests are executed, which is due to the process of stale information detection. The self-stabilizing version, which contains the mentioned functionality for detecting stale information, continuously checks the data by examining the different queues. Thus, longer queues lead to longer check procedures.

One can observe that the magnitude in execution times is similar to the results from BFTList with bounded queues, an observation that contradicts the expectation of a faster rate of increase. This is probably due to the fact that in the bounded version, the request queues have upper limits depending on the number of clients and other predefined constants in the bounded version. For example, for twelve nodes, the executed requests list can grow up to a size of $3\sigma K$, where $\sigma = 5$ is a predefined system constant and $K = 12$ is the number of clients. For the performed experiments, this yields a maximum length of 180 elements in this list. Since 204 requests were injected, each ending up in the executed requests queue eventually, this list has a similar length in both the bounded (180 elements) and the unbounded (204 elements) version during the experiments. Hence, similar execution times are not unreasonable.

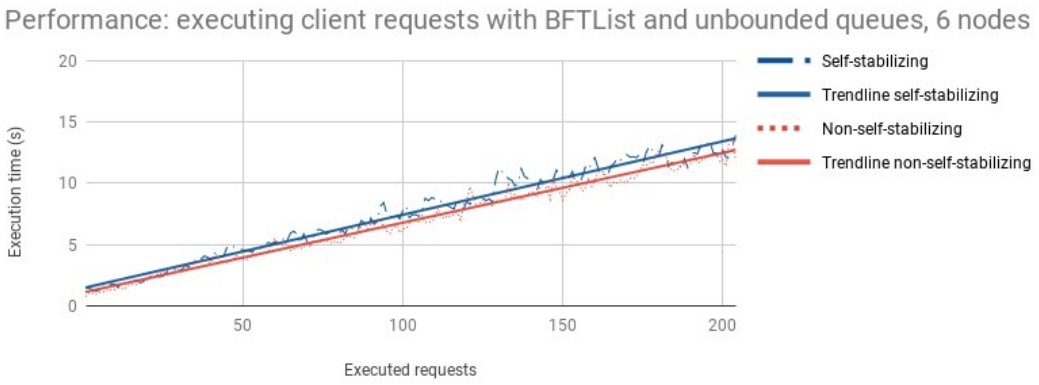

Figure 5.6: Client requests execution using BFTList with unbounded queues for six physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,0598x + 1,46$ and the lower one for non-self-stabilization with equation $y = 0,0569x + 1,11$. The trendlines grow from around two seconds to twelve seconds. The gap between the trendlines stays around below two seconds but is slightly increasing.

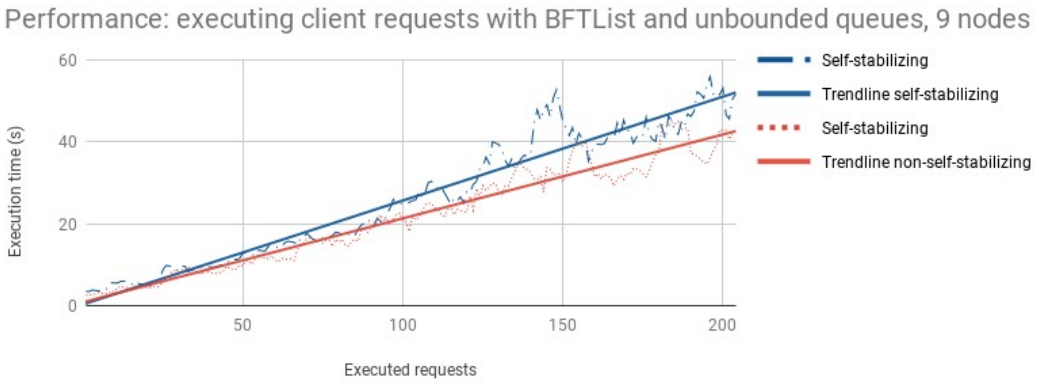

Figure 5.7: Client requests execution using BFTList with unbounded queues for nine physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,253x + 0,35$ and the lower one for non-self-stabilization with equation $y = 0,204x + 0,876$. The trendline for self-stabilizing grows from around 2 seconds to 50 seconds, while the non-self-stabilizing grows to just above 40 seconds. The gap between the trendlines is increasing up to a gap of around 10 seconds.

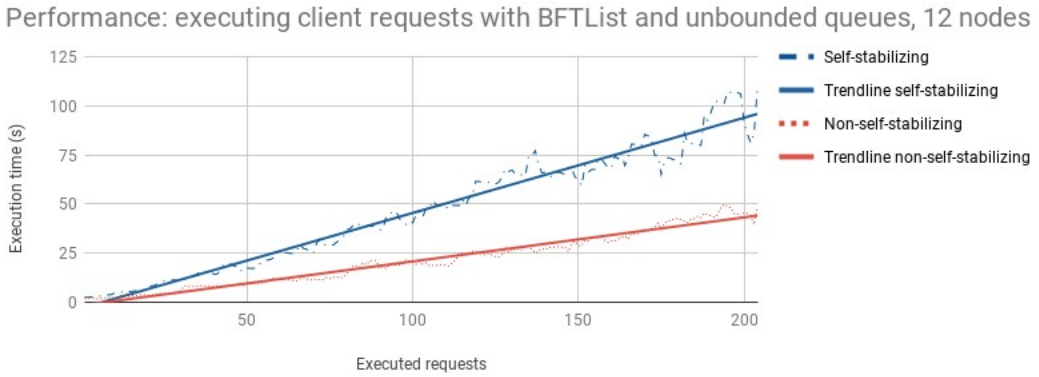

Figure 5.8: Client requests execution using BFTList with unbounded queues for twelve physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization and the lower one for non-self-stabilization. Both trendlines starts from just below zero seconds. The self-stabilizing version, with equation $y = 0,485x - 3,03$, grows to almost 100 seconds and the non-self-stabilizing, with equation $y = 0,224x - 1,6$, one to almost 50 seconds.

5.4 Performance: executing client requests with the *NO_OP* automaton

In order to investigate the overhead of self-stabilization without letting the type of state machine impact the results, experiments using the *NO_OP* automaton were conducted. Each client request contained a *NO_OP* operation, which naturally resulted in nothing being appended to the list. It is expected that executing requests containing the *NO_OP* operation requires a shorter execution time due to the removed complexity in, for example, finding stale information, compared to when using the *APPEND* operation.

The results from the client requests execution experiments using the *NO_OP* automaton can be found in Figure 5.9, Figure 5.10 and Figure 5.11 with corresponding standard deviations found in Table 5.4. The results from the two different implementations, self-stabilizing and non-self-stabilizing, of the *NO_OP* automaton can be seen in the graphs. Since the self-stabilizing version contains an additional module (the View Establishment module) and different procedures to check for stale information, it is expected that it will have longer execution times than the non-self-stabilizing version. It is also expected that the execution times will increase as the number of nodes increase.

The graphs indicate that the execution time of requests is closely related to the number of nodes in the system, since the difference in execution times for both two implementations grows with respect to the number of nodes. For six nodes, the execution time remains fairly the same as the number of injected client requests increases over time. One can see that for both versions, the execution times of the client requests are neither increasing nor decreasing but stays almost the same for client request number 0 and number 204. Furthermore, the difference between the trendlines for six physical nodes is under one second.

Both trendlines for nine nodes have a slight rate of increase with respect to the number of executed requests. The self-stabilizing version starts at around five seconds and grows to just above six seconds, while the non-self-stabilizing starts at just below four seconds and grows to around four seconds. The gap between the two trendlines therefore exhibits a minor increase.

For twelve physical nodes on the other hand, the execution times for both implementations increase significantly as the number of executed requests increases. One can see that the gap between the trendlines is slightly decreasing, which might be due to noisy data as the non-self-stabilizing version has a marginally higher standard deviation in all cases.

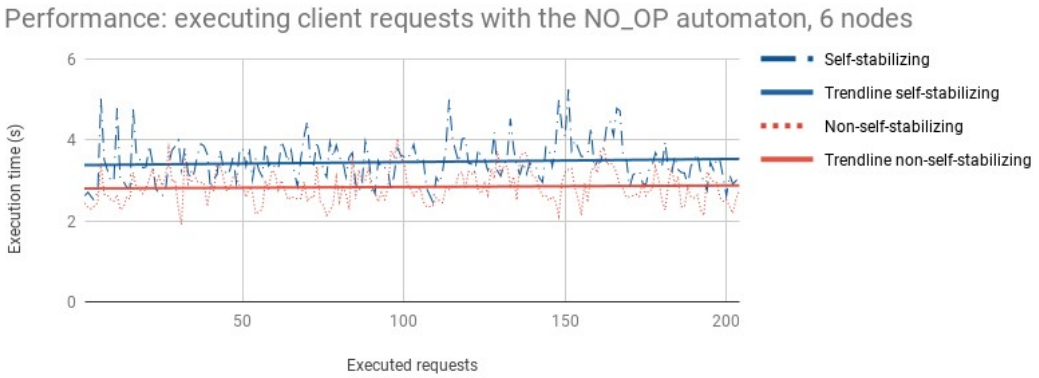

Figure 5.9: Client requests execution using the *NO_OP* automaton for six physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,000753x + 3,38$ and the lower one for non-self-stabilization with equation $y = 0,000368x + 2,8$. The trendlines are between 2,5 and 3,5 seconds.

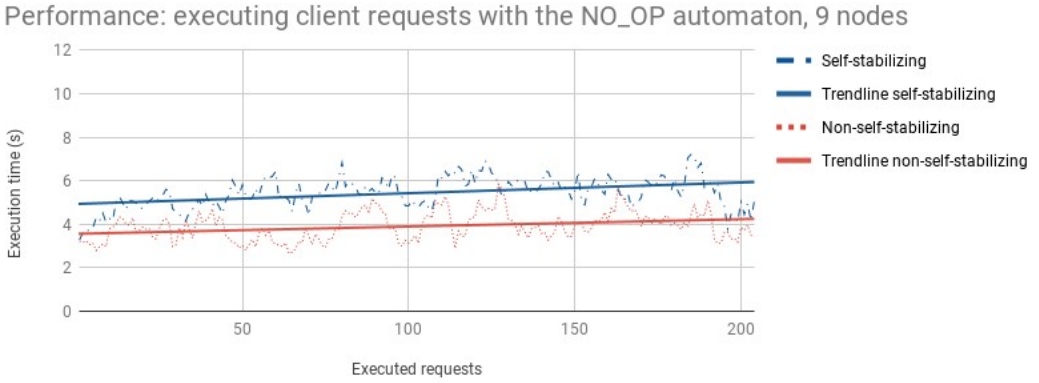

Figure 5.10: Client requests execution using the *NO_OP* automaton for nine physical nodes. The execution time of client requests can be seen on the y-axis. The upper continuous line is the trendline for self-stabilization with equation $y = 0,00493x + 4,94$ and the lower one for non-self-stabilization with equation $y = 0,0033x + 3,45$. The trendlines are between below 4 seconds and below 6 seconds.

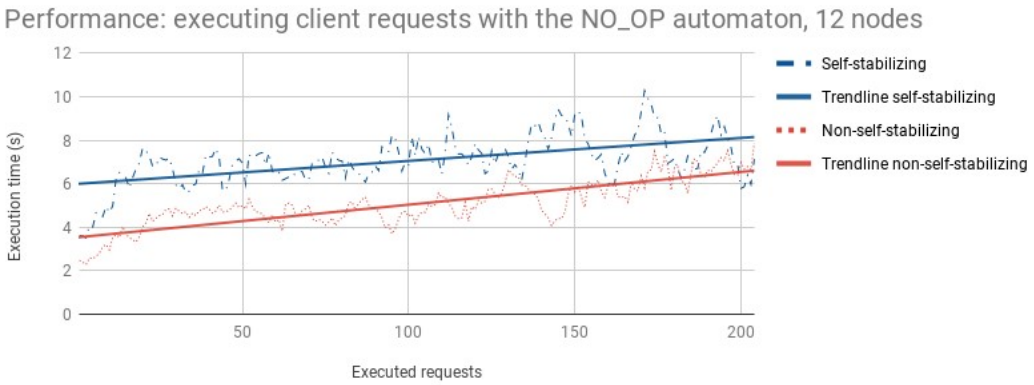

Figure 5.11: Client requests execution using the *NO_OP* automaton for twelve physical nodes. The execution time of client requests can be seen on the y-axis. The trendline for the self-stabilizing version, with equation $y = 0,0106x + 5,99$, starts at around six seconds and grows to just above eight seconds. The trendline of the non-self-stabilizing version, with equation $y = 0,0151x + 3,53$, starts at a just below four seconds and grows up to above six seconds.

Number of nodes	Version	Standard deviation
6	Self-stabilizing	13,04 %
6	Non-self-stabilizing	14,99 %
9	Self-stabilizing	13,11 %
9	Non-self-stabilizing	16,90 %
12	Self-stabilizing	13,51 %
12	Non-self-stabilizing	14,69 %

Table 5.4: The average standard deviation for each request execution time during experiments with the *NO_OP* automaton.

Comparing BFTList and the *NO_OP* automaton

Another way of visualising the request execution times is through contour plots, which can be seen in Figure 5.12 and Figure 5.13. The first figure shows how the execution times for 204 requests injected to six, nine and twelve nodes containing the APPEND operation in BFTList while the second shows the corresponding contour plot for the *NO_OP* automaton.

The BFTList plot shows the increase of execution times as both the number of nodes and the number of executed requests increase, which is an expected result. The *NO_OP* automaton plot shows how the execution time has a slight increase as the number of nodes grow, but stays fairly the same as the number of executed requests grow.

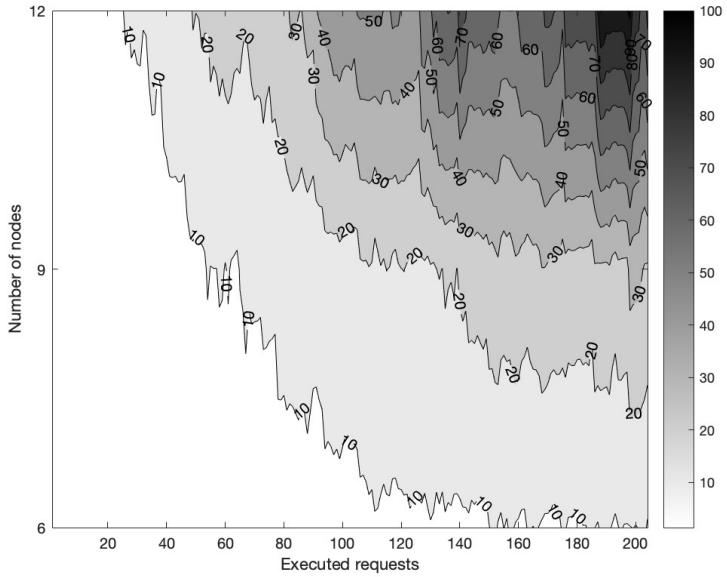

Figure 5.12: A contour plot showing the execution times for the self-stabilizing version of BFTList with six, nine and twelve nodes. The execution time grows as the number of nodes and executed client requests increase.

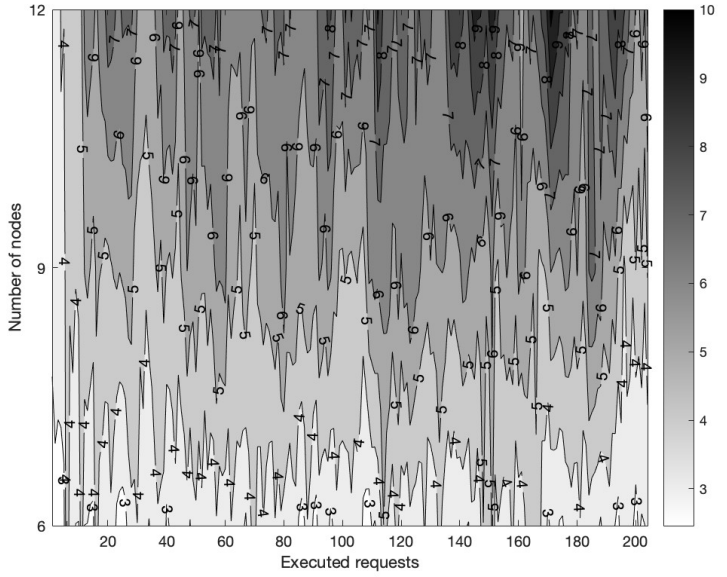

Figure 5.13: A contour plot showing the execution time for the self-stabilizing version of *NO_OP* automaton with six, nine and twelve nodes. The execution time grows as the number of nodes increase.

5.5 Performance: code profiling

In the interest of investigating the performance of BFTList even further, code profiling and experiments with respect to run times of the modules have been performed. The View Establishment module, Replication module and Primary Monitoring module all contain a do-forever loop and the run times of these functions have been measured. The time taken to prepare messages to be sent to other nodes (send method) and time spent throttling have also been measured in these modules. The experiments were conducted with BFTList on one physical node running six virtual nodes, while injecting 204 *APPEND* requests. The average execution times from three runs are presented in Table 5.5.

Module	Run method	Send method	Throttling	Iterations
VE	0,01%	98,27%	1,72%	731
R	5,45%	79,43%	15,13%	539
PM	0,03%	99,95%	0,02%	998

Table 5.5: Execution times for the do-forever loops in View Establishment (VE), Replication (R) and Primary Monitoring (PM) modules with BFTList. The run method is the actual algorithmic logic, the send method prepares messages to all other nodes and the throttling is to control the load on the communication links.

One can see that the most of the time was spent in the send method for each of the modules. Notably, the Replication module spent a significant longer amount of time throttling than the two other modules. This is probably due to the fact that as the list grows, processing an incoming message takes longer and longer time. This forces the sending nodes to throttle, to not overload the communication links.

The same experiments as above were conducted using the *NO_OP* automaton. The resulting execution times can be found in Table 5.6. The percentage of time spent throttling for the Replication module is now heavily reduced, due to the fact that the list was no longer constantly growing and therefore the message size was not increasing in the same fashion. The two other modules have a similar distribution of time as the results from BFTList. This is expected since the *NO_OP* automaton is only affecting the performance of the Replication module. This result strengthens the hypothesis that the additional throttling time when running BFTList is due to the overhead of having an increasing size of the list. The same amount of iterations was performed for each of the modules in the two different implementations; BFTList and *NO_OP* automaton.

In order to identify key methods and functions with the longest execution times, an additional profiling of the modules was performed. The experiments were done locally with BFTList due to the overhead of conducting such profiling. The profiling tool Yappi slows down the system dramatically and therefore the experiments were run locally using six virtual nodes and only twelve requests with *APPEND*

Module	Run method	Send method	Throttling	Iterations
VE	0,10%	98,13%	1,77%	756
R	1,32%	95,69%	2,99%	521
PM	0,32%	99,51%	0,17%	1029

Table 5.6: Execution time for the do-forever loops in View Establishment (VE), Replication (R) and Primary Monitoring (PM) modules with the *NO_OP* automaton. The run method is the actual algorithmic logic, the send method is preparing messages to all other nodes and the throttling is to control the load on the communication links.

operations were injected. The results show that the receiving and sending of messages have long execution times compared to the other functions. When it comes to longest average execution time, the sending method in the Replication module has the longest execution time and the longest total execution time is spent in the receiving method in the same module. It should be noted that the sending method comes in second with respect to total execution time during experiments. The fact that methods related to communication is the top two longest executing functions is not surprising due to the fact that client requests are being continuously injected. Having *APPEND* operations puts a high workload on the replication messages since the complete current list is being attached to each message. Therefore, a long execution time on both the sending method and receiving method are expected.

When disregarding the methods related to sending and receiving messages, the View Establishment method "get_current_view" is responsible for the longest total execution time. This method returns the current view of the node and is called from all modules several times. The longest average execution time belongs to a method in the Replication module which returns client request which has been seen at at least $n - 2f$ nodes. This function is called several times during different phases in the 3-phase commit protocol. In order to retrieve all requests, the method needs to search through different request queues for all other nodes, hence a long execution time is expected.

The code profiling experiments together indicate that the system spent most time at inter-node communication. When client requests are executed the Replication module has the heaviest workload with respect to communication, as expected.

5.6 Performance: convergence time

These experiments were conducted using BFTList but *without* injection of client requests. The main focus was to investigate how long it takes for the system to converge to a stable view in the presence of an unresponsive primary, which is referred to as the *convergence time* of the system. In order to conduct a view change and establish a new view, $n - f$ processors need to agree before doing so and the convergence time is considered to be the time taken for this agreement to be per-

formed. This time is of high importance for the self-stabilizing BFTList, as during this period no client requests will be executed. It is necessary for the system to be in a stable view in order to execute requests and it is therefore desirable to have a short convergence time. However, as $n - f$ nodes need to agree before moving to the next view, more nodes in the system suggests a longer convergence time. It is therefore expected that the convergence time will increase as more nodes are added to the system.

The convergence time results can be found in Table 5.7, where the results from experiments both with and without the event-driven failure detector are presented. The numbers in percentage are the standard deviations. Experiments conducted with the event-driven failure detector show a significantly longer convergence time than the system without it. This is due to the additional agreement protocol used in the event-driven failure detector and was expected.

When the number of Byzantine nodes is kept at minimum, $f = 1$, the convergence time increases as more nodes are added to the system. For $f = 1$ all nodes, except the Byzantine node, are required to cooperate when establishing a stable view. Therefore, an increasing convergence time is expected. However, when maximising the allowed number of Byzantine nodes, the results show a significantly faster convergence time for twelve nodes (last row in Table 5.7). This is most likely due to the fact that fewer nodes need to agree in order to establish a new view. For example, when $n = 12$ and $f = 2$, only ten nodes $(n - f)$ need to agree before establishing a stable view. This also means that all agreement steps leading up to a stable view, also require a slightly less amount of nodes in comparison with $n = 12$ and $f = 1$. Another possible reason for the somewhat surprising results can be explained by the fact that the convergence time is calculated from the $n - f$ fastest nodes. Experiments on PlanetLab showed that the nodes being **excluded** for the view establishment for twelve physical nodes are in fact **included** for the view establishment for nine physical nodes. This means that some nodes during the experiments with nine physical nodes are "slower" than the additional nodes used during the experiments with twelve physical nodes, hence these "slow" nodes are part of the explanation for a longer convergence time. This conjecture is supported by the results from experiments conducted in a local environment, found in Table 5.8. When running locally (nodes emulated as separate processes on the same machine), all virtual nodes have similar settings and the system is more homogeneous than when running on PlanetLab. Experiments were only conducted with the main implementation, meaning that the system did not contain the event-driven failure detector. Furthermore, the same experiment was performed using the cloud-provider AWS[1] yielding similar results as in the local environment. The results show that the convergence time for nine physical nodes are indeed faster than for twelve physical nodes regardless the number of Byzantine nodes, supporting our hypothesis.

The corresponding standard deviations for the experiments on PlanetLab can be found in Table 5.7 and for experiments conducted locally in Table 5.8. One can

[1] https://aws.amazon.com/

see that the standard deviation varies a lot no matter the number of nodes in the system. For six nodes without the event-driven failure detector the standard deviation is significantly higher than for nine and twelve nodes. When the failure detector is enabled, the result with twelve nodes (with one Byzantine node) has the highest standard deviation. This unpredictable nature of these deviations indicate that the results are dependent on the system state and system load at the time of running the experiments. For example, since the convergence time is relatively short for six nodes (below three seconds) and five nodes are needed to establish a view ($n - f = 6 - 1 = 5$), one node with a strongly varying communication latency can yield a high deviation in the experiments. The convergence time is longer for nine and twelve nodes, meaning that a deviation in one node does not necessarily effect the total convergence time to the same extent. On the other hand, when having a system with twelve nodes, the risk of having one or several "slow" nodes increases since the probability of having to wait for one slow node is higher. This might also yield a high deviation. The experiments run locally show the highest deviation for nine nodes, once again yielding an unpredictable behaviour of standard deviations.

No. of nodes	**No. of Byzantine nodes**	**Without ED-FD**		**With ED-FD**	
6	1	2,85 s	(17,99%)	19,18 s	(4,54%)
9	1	12,49 s	(1,74%)	19,57 s	(10,36%)
12	1	13,67 s	(4,37%)	20,75 s	(17,22%)
12	2	5,92 s	(3,32%)	14,82 s	(9,86%)

Table 5.7: The convergence times for conducting a view change with six, nine and twelve nodes on PlanetLab with the corresponding standard deviations shown in parenthesis. Two different settings were conducted with twelve nodes, one with $f = 1$ and one with the maximal number of Byzantine nodes allowed $f = 2$.

No. of nodes	**No. of Byzantine nodes**	**Without ED-FD**	
6	1	1,57 s	(8,79%)
9	1	3,37 s	(27,09%)
12	1	8,93 s	(10,46%)
12	2	6,48 s	(2,67%)

Table 5.8: The convergence times for conducting a view change with six, nine and twelve virtual nodes in a local environment. Two different settings were conducted with twelve nodes, one with $f = 1$ and one with the maximal number of Byzantine nodes allowed $f = 2$.

6
Discussion

This chapter presents a summary of the project, discusses possible extensions and lessons learned from validating and evaluating a self-stabilizing distributed system. We round off this chapter with a conclusion of the project.

In this project the SSPBFT algorithm by Dolev *et al.* [3] has been practically validated showing correctness of the proof invariants. The algorithm has been further validated and evaluated in a real network, the PlanetLab EU platform. The results indicate that the SSPBFT algorithm performs best with bounded state machines. Many modules of the system may also be useful independently for other systems implementing distributed algorithms.

6.1 Extensions

Communication in a distributed algorithm is non-trivial to implement, while at the same time it is something that just needs to "be there" in the sense that it is often considered out of scope for algorithm designers. It is simply assumed to be there and function in the expected way. Because of this, the communication layer in this project was implemented in a way that facilitated rapid development. This had some drawbacks, such as the fact that the communication module did not scale very well. One extension which could be worthwhile to investigate in order to optimise the communication could be to use a compression algorithm for the messages. Examples of libraries of such compression algorithms developed for Python are lzma[1], zlib[2] and bz2[3]. We implemented a proof-of-concept for this type of optimisation, using the library *lzma*. Even though it worked to compress the messages, it showed no significant improvement. There is a trade-off from where a compression algorithm will give an improvement depending on the size of the messages and the overhead of conducting the compression. This can potentially be investigated further in an optimised version.

Another potential extension could be to reduce the number of messages sent by each module. For example, a module could decide not to send a messages if the content is the same as in the previous message sent. For extra caution, the modules could send the messages every now and then, even if the content is the same, to make sure

[1] https://docs.python.org/3.4/library/lzma.html
[2] http://www.zlib.net/
[3] https://docs.python.org/3/library/bz2.html

that information-sharing is still active in the system.

The studied algorithm can serve as the basis for cloud systems [8, 7]. We note the existence of complementary self-stabilizing IT infrastructure, such as self-stabilizing [24, 25] and quorum reconfiguration [26]. Furthermore, other extensions to the studied algorithm are both models for provable rationalities [27, 28, 29, 30] that could be based on BAR [31] as well as the coding and preservation of privacy for state machines modelling emulated state [32, 33].

The implemented state machine was designed to be a validation (proof of concept) of the SSPBFT algorithm, rather than providing a foundation for easy self-stabilizing overhead measurement. The unbounded BFTList state machine kept growing, as request operations only appended to the list, and due to the exchange between nodes of each current state this resulted in large messages. Messages continuously growing in size affected the client requests execution result in a manner that was dependent on the type of state machine. The solution to enable a more fair comparison regarding the self-stabilizing overhead was to use the "no operation" request type in the *NO_OP* automaton. This "simple" fix could have been avoided by designing the system with another type of state in mind, for example a vector clock or a simple counter. A simple state machine would reduce the complexity of logic connected to the type of state machine but still give a good indicator of overhead, performance and scalability. However, our system when using the append-request type for the state machine provides a more advanced state machine that can be compared to a blockchain.

We believe that the validation of the algorithm by Dolev *et al.* can pave the way for other projects, such as the bootstrapping of self-stabilizing blockchain. Furthermore, due to the module-based approach, the codebase and architecture can easily be used in other projects implementing self-stabilizing distributed algorithms.

6.2 Validating and evaluating a self-stabilizing distributed system

Systems that require inter-node communication also require more complex testing than just unit testing. Such tests are often non-trivial to implement and there is also the question of how the system will behave in production versus when running in a development environment. Another issue one needs to consider is if the tests are testing the correct functionality or not, since there are no "global" tests that validate the tests. In our case there is also the matter of testing that the system stays in its legal execution, meaning for example that a state stays consistent or that a view stays stable, according to the closure property of self-stabilization. How much time is "enough" time to be certain that the system remains in its legal execution? We can only test the system within a finite time interval and only claim correctness for these cases. In the same way as not all possible cases can be tested and implemented, running the system for a period of time that can be considered

as an infinity is not feasible.

Applications deployed to many servers often come with a need to monitor these systems, in order to understand performance, bottlenecks and various other things. Metrics gathering and visualisation are tools often used to do so, but they come with an inherent problem. What if the constructed graphs are wrong? There are no tests that can be written and since the nature of the underlying distributed system emitting these metrics is highly complex, there is simply no way to know if a graph is correct. The possibility of graphs being wrong will always be present and needs to be taken into consideration when using these types of graphs for evaluation of the results.

6.3 Conclusion

In this project a distributed system has been implemented in order to practically validate the SSPBFT algorithm by Dolev *et al.* [3]. The algorithm is confirmed to be functioning correctly with respect to the implemented invariants found in the theoretical proofs in [3]. The preliminary evaluation shows that the performance of the system is related to both the number of nodes in the system and also the size of the state machine. The type of state machine used is of high importance and using a constant size state machine has been shown to be able to reduce the execution time of requests up to 10x compared to when using an unbounded state machine. The existing codebase can also easily be extended to be used for other distributed systems or other distributed self-stabilizing algorithms.

Bibliography

[1] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. In *ACM Trans. Comput. Syst*, volume 20, pages 398–461, 2002.

[2] EW Dijkstra. Self-stabilizing systems in spite of distributed control. In *Communications of the ACM*, volume 17, pages 643–644, 1974.

[3] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing byzantine tolerant replicated state machine based on failure detectors. In *Proceedings of CSCML and technical report*, pages 84–100, 2018.

[4] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *ACM Trans. Comput. Syst.*, volume 27, pages 1–39, 2009.

[5] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. In *ACM Trans. Comput. Syst.*, volume 32, pages 1–45, 2015.

[6] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[7] 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 *SSS*, volume 10083 of *Lecture Notes in Computer Science*, pages 36–53, 2016.

[8] Ariel Daliot and Danny Dolev. Self-stabilization of byzantine protocols. In *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Proceedings*, pages 48–67, 2005.

[9] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Practically-self-stabilizing virtual synchrony. In *J. Comput. Syst. Sci.*, volume 96, pages 50–73, 2018.

[10] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. In *IEEE Trans. Mob. Comput.*, volume 5, pages 893–905, 2006.

[11] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. In *Acta Inf.*, volume 40, pages 609–636, 2004.

[12] Shlomi Dolev and Elad Schiller. Communication adaptive self-stabilizing group membership service. In *IEEE Trans. Parallel Distrib. Syst.*, volume 14, pages 709–720, 2003.

[13] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. In *J. Comput. Syst. Sci.*, volume 76, pages 884–900, 2010.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, 1978.

[15] Mike Paterson Michael J. Fischer, Nancy A. Lynch. Impossibility of distributed consensus with one faulty process. In *PODS*, pages 1–7, 1983.

[16] Robert Shostak Leslie Lamport and Marshall Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 382–401, 1982.

[17] Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *The Distributed Computing Column*, pages 2–23, 2017.

[18] Synnöve Kekkonen-Moneta Joffroy Beauquier. Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. In *International Journal of Systems Science*, pages 1177–1187, 1997.

[19] Mikhail Nesterenko Sebastien Tixeuil Swan Dubois, Maria Potop-Butucaru. Self-stabilizing byzantine asynchronous unison. In *J. Parallel Distrib. Comput. 72*, pages 917–923, 2012.

[20] Shlomi Dolev. *Self-Stabilization.* The MIT Press, 2000.

[21] Iosif Salem and Elad M. Schiller. Practically-self-stabilizing vector clocks in the absence of execution fairness. *Podelski A., Taïani F. (eds) Networked Systems*, pages 318–333, 2018.

[22] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In *SSS*, volume 7596 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2012.

[23] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. Self-stabilizing byzantine resilient topology discovery and message delivery - (extended abstract). In *NETYS*, volume 7853 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2013.

[24] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. A self-organizing distributed and in-band SDN control plane. In *ICDCS*, pages 2656–2657. IEEE Computer Society, 2017.

[25] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. Renaissance: A self-stabilizing distributed SDN control plane. In *ICDCS*, pages 233–243. IEEE Computer Society, 2018.

[26] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing reconfiguration. In *NETYS*, volume 10299 of *Lecture Notes in Computer Science*, pages 51–68, 2017.

[27] Shlomi Dolev, Panagiota N. Panagopoulou, Mikaël Rabie, Elad Michael Schiller, and Paul G. Spirakis. Rationality authority for provable rational behavior. In *Algorithms, Probability, Networks, and Games*, volume 9295 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2015.

[28] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Robust and scalable middleware for selfish-computer systems. In *Computer Science Review*, volume 5, pages 69–84, 2011.

[29] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Strategies for repeated games with subsystem takeovers implementable by deterministic and self-stabilising automata. In *IJAACS*, volume 4, pages 4–38, 2011.

[30] Shlomi Dolev, Panagiota N. Panagopoulou, Mikaël Rabie, Elad Michael Schiller, and Paul G. Spirakis. Rationality authority for provable rational behavior. In *PODC*, pages 289–290. ACM, 2011.

[31] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 45–58. ACM, 2005.

[32] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. Brief announcement: Robust and private distributed shared atomic memory in message passing networks. In *PODC*, pages 311–313. ACM, 2015.

[33] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. *CoRR*, abs/1806.03498:arXiv:1806.03498, 2018.

A

Application threads

A list of all threads launched by BFTList and their purpose can be seen in Table A.1. The number of threads, T, for the application can be calculated as $T = n + 9 + \lambda$, where n is the number of nodes in the system and λ is the amount of timeout threads launched by the failure detector at the time of measuring. Further details of the communication protocol using this timeout mechanism can be found in Section 3.2.3.3.

Thread name	Purpose	Count
Main thread	Entrypoint of application	1
View Establishment module	Runs the module in a separate thread	1
Replication module	Runs the module in a separate thread	1
Primary Monitoring module	Runs the module in a separate thread	1
Failure Detector module	Runs the module in a separate thread	1
Event-driven Failure Detector	Not used in the main implementation	(1)
Prometheus metrics endpoint	Exposes application metrics	1
Web API	RPC and global view	1
Latency monitor	Monitors latency to other nodes	1
TCP Receiver	Receives messages over TCP	1
UDP Receiver	Receives messages over UDP	1
UDP Sender	Sends messages over UDP	$n - 1$
UDP Sender Timeout	Timeout mechanism for UDP senders	λ

Table A.1: An extensive list of all threads launched in the application and their usage. Summing up the total number of threads for the main application, T, can be done as $T = 10 + (n - 1) + \lambda = n + 9 + \lambda$. Since the number of timeout threads vary throughout the execution of the system, λ is used to represent this number. Note that there is no TCP Sender in the table, which is due to this type of sender being constructed using event-driven programming rather than threads. Note that the event-driven failure detector is not part of the thread count formula, since it is not present in the main implementation.

B

Communication algorithms

B.1 Sender channel algorithm

Algorithm 5 Algorithm for the sender channel

```
1:  upon timeout
2:      send(counter)
3:  upon message arrival
4:  begin
5:      receive(MsgCounter)
6:      if MsgCounter ≥ counter then
7:          begin
8:              counter := MsgCounter + 1
9:              send(counter)
10:         end
11:     else
12:         send(counter)
13: end
```

B.2 Receiver channel algorithm

Algorithm 6 Algorithm for the receiver channel

```
1:  upon message arrival
2:  begin
3:      receive(MsgCounter)
4:      if MsgCounter ≠ counter then
5:              counter := MsgCounter
6:      send(counter)
7:  end
```

C

Invariants of the proofs

The following notation is used to present the invariants.

- c is a system configuration
- e is an execution fragment starting in c, representing actions performed by processors leading to a new system configuration c'
- E is an execution, meaning an execution (containing execution fragments) of the algorithm starting in c and ending in another c'

Each invariant is presented with a starting system configuration, c_s, and the desired system configuration, c_d, to end in. The system should finish in c_d, in order to validate the invariant. It should be noted that the notation of execution fragment (e) indicates that there can be several steps of the algorithm executed leading up to the desired system configuration (c_d). The numbering of lemmas, claims and theorems in this appendix corresponds to the numbering in [3] and the invariants are grouped with respect to the modules of the implemented algorithm.

C.1 View Establishment module

Lemma 1- Processors start in different *views*, hence the system is in an arbitrary state.

c_s: The processors have distinct current views due to stale information, hence there is no stable view.

c_d: There exists no stale information at any correct processor.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will ensure that all stale information has been removed, e.g. all stale variables are reset.

Testing: The above invariant was tested by having the processors start executing the algorithm with different predefined views assigned as their correct current view. All processors were acting correctly. This was tested in a distributed manner since exchange of information was necessary to detect the inconsistency in views, hence the stale information.
Validating: No stale information existed at any correct processor.

Lemma 9 - Byzantine processor sends different current view to different processors and halts the progression.

c_s: A processor p_i is acting Byzantine.

c_d: No progression is made in the system.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will **not** lead to a safe configuration because the Byzantine behaviour successfully halts the procedure of establishing a stable view.

Testing: The above invariant was tested by defining the starting configuration state of views according to appendix in [3] and also emulating the Byzantine behaviour of sending different views to different processors at one processor. All processors except p_i were acting correctly. This was tested in a distributed manner since communication was needed to emulate the Byzantine behaviour.
Validating: No progression was made in establishing a stable view, hence the consistency set describe in Lemma 9 in [3] is necessary for progression.

Theorem 10- Convergence, a stable view is eventually established.
c_s: The system starts in an arbitrary state.
c_d: A stable view is established in the system.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that all correct processors have the stable view as their current view.

Testing: The above invariant was tested by having the processors start in an arbitrary manner, meaning they could be in different starting views. This was tested in a distributed manner since communication was needed to agree upon a view.
Validating: All correct processors had the same, stable view as their current view.

Theorem 11- Closure, the system remains in a stable view.
c_s: The system starts in an stable view.
c_d: The system remain in a stable view.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that there exists a stable view in the system.

Testing: The above invariant was tested by having the system start in a stable view, meaning that correct processors started in the same current view. This was tested in a distributed manner since communication was needed to possibly agree upon a new view.
Validating: All correct processor had the same, stable view as their current view.

C.2 Replication module

Claim 12 - Processor starts with stale information regarding the replication leading to a reset of the Replication module.

c_s: Processor p_i has stale information in the Replication module.
c_d: No stale information exists at p_i.

The execution containing the above configurations and an execution fragment $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that all stale information is removed and a reset of the Replication module has occurred at the processor.

Testing: The above invariant was tested by having the processor start executing the algorithm with stale information, such as an sequence number higher than the threshold or/and having the same request with different sequence numbers. The processor was acting correctly. The invariant could be tested locally (using only one processor), since stale information could be detected by a single processor by examining its own data.
Validating: A call to the reset method was conducted and the stale information was removed from the processor.

Claim 14 (and Lemma 17)- More than $2f + 1$ processors start with non-consistent replica states.
c_s: More than $2f + 1$ processors have distinct replica states.
c_d: More than $2f + 1$ have a consistent (default) replica state.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that a correct consistent replica state exists at at least $2f+1$ replicas. This means that the other $(n-(2f+1)-f = 2f)$ correct processors cannot find a consistent replica state that differs from the default state ($2f$ nodes does not provide enough support for such a state). This set of correct processors will therefore also assign the default state as their replica state. The state machine is thus consistent throughout the replicas.

Testing: The above invariant was tested by having more than $2f + 1$ processors start executing the algorithm with distinct replica states. This was tested in a distributed manner since the processors needed to exchange information in order to detect the inconsistency and eventually assign the default replica state.
Validating: The replica states of at least $3f + 1$ processors were changed to the default state.

Lemma 19- One processor has executed an unsupported client request, the processor reassign its replica state to the consistent replica state found in the system.
c_s: Processor p_i has executed an unsupported request hence its replica state differs from the other processors.
c_d: The replica state of p_i is coherent with the majority of the other processors.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that the faulty replica structure at p_i is removed. The state machine is thus consistent throughout the replicas.

Testing: The above invariant was tested by having processor p_i start executing the algorithm with an unsupported client request already executed. All processors were acting correctly. This was tested in a distributed manner since the processor needed to exchange information in order to find the correct consistent replica state.
Validating: The replica state of processor p_i was changed to the consistent state across system.

Lemma 20- At least $3f + 1$ processors have a prefix of the correct replica state, all correct processors will eventually adopt the correct replica state.

c_s: $3f + 1$ processors has a prefix of a predefined correct replica state.

c_d: All correct processors will either have the correct replica state or a prefix, hence converging to the correct state.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that the state machine is consistent across the correct processors.

Testing: The above invariant was tested by letting $3f+1$ processors start executing the algorithm with a prefix of a predefined replica state, which was considered to be the correct current replica state. This was tested in a distributed manner since communication was necessary for finding the current consistent replica state.
Validating: All correct processors had the predefined correct replica state or a prefix of the replica state.

Lemma 20[1]- At least $2f + 1$ but less than $3f$ processors have a prefix of the correct replica state, all correct processors will eventually adopt the correct replica state.

c_s: $2f + 1$ processors have a prefix of a predefined correct replica state.

c_d: All correct processors will either have the correct replica state or a prefix, hence converging to the correct state.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense the state machine is consistent across the correct processors.

Testing: The above invariant was tested by letting $2f+1$ processors start executing the algorithm with a prefix of a predefined replica state, which was considered to be the correct current replica state. This was tested in a distributed manner since communication was necessary for finding the current consistent replica state.
Validating: All correct processors had the predefined correct replica state or a prefix of it.

Claim 24- The primary is acting Byzantine regarding the assignment of sequence numbers.

[1]Two cases of Lemma 20

c_s: The current primary is acting Byzantine and the system is in a safe state with a stable view.
c_d: The system is not effected by the Byzantine primary, meaning that the system remains in a safe state.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration state in the sense that the malicious behaviour of the Byzantine primary has not successfully damaged the consistency in the system regarding the replica state or view.

There are four different Byzantine behaviours that a primary can exhibit while assigning sequences numbers; reuse sequence numbers, stop assigning sequence numbers, assign sequence numbers out of the given threshold or assign different sequence numbers to the same client request. All these behaviours have been tested and validated.

Testing: The above invariant was tested by emulating the Byzantine behaviour at the starting primary processor. The algorithm was executed and incoming client requests were emulated during the execution. This was tested in a distributed manner since communication was needed in order to, for example, confirm the assignment of sequence numbers.
Validating: The systems safe state had not been violated by the Byzantine primary, meaning that the primary had not managed to break the system.

Claim 25- A view change has occurred, leading to a correct primary and there are pending client requests waiting to be executed.

c_s: A new correct primary processor has just been elected, there exists pending (not executed) client requests at the processors.
c_d: The pending client requests have been executed and the state machine is consistent, including a stable view, across the system.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that a stable view is established and the state machine is consistent at the replicas.

Testing: The above invariant was tested by having the processors start executing the algorithm with a predefined primary and with pending requests already in their pending queues. All processors were acting correctly. This was tested in a distributed manner since communication was necessary to conduct the 3-phase commit protocol.
Validating: The pending client requests were executed and the system was in the stable view that was predefined.

C.3 Primary Monitoring module

Lemma 26- No more than one correct primary should be falsely suspected due to faulty information

c_s: No primary is suspected.

c_d: A new correct primary has been elected and since a reset has been conducted the variables are re-initialised hence no stale information exist. The correct primary is thus not suspected.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that a new correct primary is elected. During the execution the total number of primaries suspected is at maximum one. Hence no more than one correct primary has been suspected due to the stale information.

Testing: The above invariant was tested by having the different processors start executing the algorithm with different type of stale information. For example, some processor had data indicating that the primary is unresponsive and others had data indicating that the primary is not making progress. Stale information like that lead to a suspicion of the primary at a processor level. All processors were acting correctly. This was tested in a distributed manner since the processor needed to exchange information in order to conduct a view change.
Validating: A new primary was elected and was not suspected due to the faulty start data, since the data was removed during the reset of the algorithm at the view change.

Lemma 29 - A non-faulty primary should not be suspected.

c_s: Processor p_i is the current primary processor.

c_d: p_i is still the primary processor.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will remain in a safe configuration in the sense that there exists a stable view, hence a primary, and no processor has suspected the correct primary p_i as a corrupt processor.

Testing: The above invariant was tested by having the system start in a stable view and emulating incoming client requests. All processors were acting correctly. This was tested in a distributed manner since information need to be exchanged to confirm responsiveness of the primary and progression of the state machine.
Validating: No suspicion was detected at any processor and the view, hence the primary, remains the same.

Lemma 29[2]- Byzantine primary becomes unresponsive.

c_s: The primary processor p_i is acting Byzantine.

c_d: The replica states are consistent across the system.

[2]Two cases of Lemma 29

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that there exists a stable view and the state machine is consistent across the system.

Testing: The above invariant was tested by emulating the Byzantine behaviour of being unresponsive at the predefined primary processor. All processors were acting correctly except the primary processor which was unresponsive. This was tested in a distributed manner since communication was necessary to detect unresponsiveness.
Validating: A view change had occurred and a new stable view was established. Any incoming client requests were executed, meaning that the progression of the state machine was not affected and the state machine was consistent.

C.4 Mal-free executions

Invariant that does not correspond to a specific lemma or claim.

The system is acting correctly and client requests are valid, thus executed.

c_s:	The algorithm is initialised at all processors.
c_d	The state machine is consistent across the system and the view is stable.

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that there exists a stable view and a consistent replica state in the system.

Testing: The above invariant was tested by running BFTList at a number of processors and emulating incoming client requests. All processors were acting correctly. This was tested in a distributed manner since communication was necessary to establish views and conduct the 3-phase commit protocol.
Validating: The incoming client requests were executed in a coherent manner, hence the state machine was consistent in the system and the system was in a stable view.

f processors are more than $3\sigma K$ executed requests behind the current correct replica state[3].

c_s:	The last executed request at f processors has a sequence number with a difference of more than $3\sigma K$ from the last common executed sequence number.
c_d	The sequence number of the last executed request at the f processors is consistent with the sequence number of the last common executed request.

[3] σ is a predefined system constant

The execution containing the above configurations and an execution fragment, $E = (c_s, e, c_d)$ will lead to a safe configuration in the sense that there is a consistent replica state in the system and the processors that were behind have catch up.

Testing: The above invariant was tested by having one processor being behind the rest, with respect to executed sequence numbers. This was tested in a distributed manner since communication was necessary to discover that the processor was behind and to exchange the current state.
Validating: The processor that was behind had a replica state that was consistent with the other processors.

D

PlanetLab nodes

The physical PlanetLab nodes used for evaluation. As can be seen in the table below, the nodes are fairly distributed in different physical locations.

Hostname	**TLD**	**IP**
planetlab1.upm.ro	ro	193.226.19.30
planetlab-1.ida.liu.se	se	192.36.94.2
planetlab2.xeno.cl.cam.ac.uk	uk	128.232.103.202
planetlabeu-2.tssg.org	org	193.1.201.27
planetlabeu-1.tssg.org	org	193.1.201.26
ple4.planet-lab.eu	eu	132.227.123.14
planetlab-2.cs.ucy.ac.cy	cy	194.42.17.164
ple44.planet-lab.eu	eu	132.227.123.44
planet4.cs.huji.ac.il	il	132.65.240.103
cse-white.cse.chalmers.se	se	129.16.20.71
cse-yellow.cse.chalmers.se	se	129.16.20.70
mars.planetlab.haw-hamburg.de	de	141.22.213.35