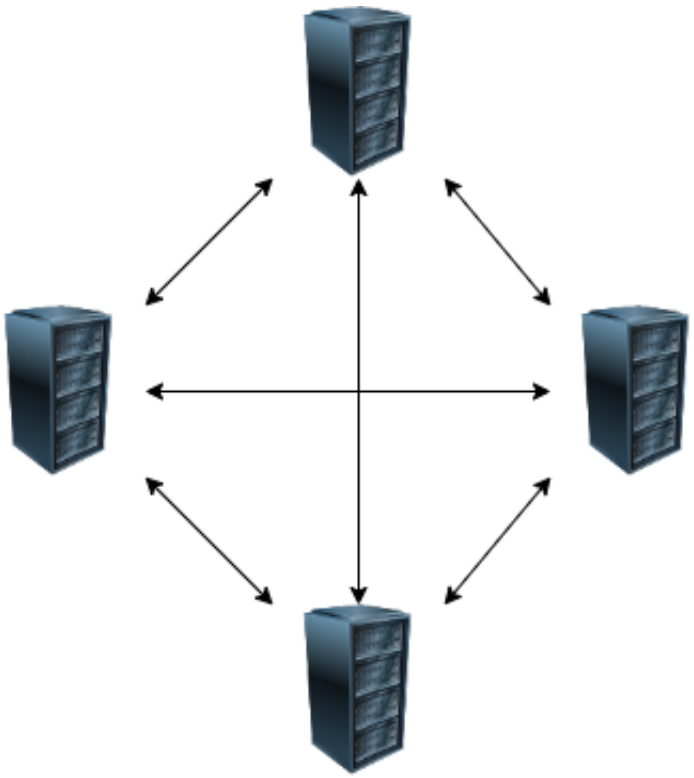

Self-Stabilizing Emulation of State-Machine Replication

Implementation and Evaluation of Self-Stabilizing Emulation of State-Machine Replication with Global Restart Mechanism

Master's thesis in Computer Systems & Networks

Daniel Karlberg
Daniel Kem

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Self-Stabilizing Emulation of State-Machine Replication

An Implementation and Evaluation of Self-Stabilizing Emulation of State-Machine Replication with Global Restart Mechanism

Daniel Karlberg
Daniel Kem

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science & Engineering
Division of Networks & Distributed Systems
Distributed Computing and Systems
Chalmers University of Technology
Gothenburg, Sweden 2021

Self-Stabilizing Emulation of State-Machine Replication
An Implementation and Evaluation of Self-Stabilizing Emulation of State-Machine Replication with Global Restart Mechanism
Daniel Karlberg
Daniel Kem

Supervisors: Elad Michael Schiller, Department of Computer Science & Engineering
Robert Gustafsson, Combitech AB
Ingvar Andersson, Combitech AB

Examiner: Tomas Olovsson, Department of Computer Science & Engineering

Master's Thesis 2021
Department of Computer Science & Engineering
Division of Networks & Distributed Systems
Distributed Computing and Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Communicating servers, authors' own illustration.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Self-Stabilizing Emulation of State-Machine Replication

Daniel Karlberg
Daniel Kem
Department of Computer Science & Engineering
Chalmers University of Technology

Abstract

The use of distributed systems has grown dramatically in our modern society over the past few decades. Today we see many digital services being distributed such as cloud storage and automotive systems. These distributed services are being used by every sector of our society from banking and finance to more traditional industries such as manufacturing and military. In other words, These systems are part of our daily lives and are vital to the daily operation of our society. At the same time as these systems are important, they also create critical points in our society. Faults and failures of these distributed systems could have large negative effects on fragile parts of society.

This is the reason for the initialization of this project, to implement and validate recent research and advancements in the area of self-stabilization and fault-tolerance for distributed systems, and evaluate if the research can increase the levels of fault-tolerance in these systems. This research could allow distributed systems to not only handle communication faults and crashing of participants in the system, but also arbitrary transient faults.

Our project has contributed with the first, the best to our knowledge, real-world implementation and evaluation of a stack of self-stabilizing distributed algorithms, which together provide the service of a self-stabilizing emulation of state-machine replication. The project demonstrates that recent advancements in the area of self-stabilization can provide real-world distributed systems with a higher degree of fault-tolerance.

Keywords: self-stabilizing, self-stabilization, distributed systems, distributed computing, fault-tolerance, consensus, reconfiguration, state-machine, replication, total-order broadcast.

Acknowledgements

We want to thank our Chalmers supervisor Elad Michael Schiller, who worked closely with us and provided unwavering support, guidance and expertise to us whenever we needed it. Always being there with a quick message away from having a meeting about our development.

We also want to thank our industrial supervisors from Combitech AB, Ingvar Andersson and Robert Gustafsson for their important input and guidance from the industry during our project.

Our thanks also extend to Amanda Sjöö, who worked on a parallel project and helped us find bugs during our development.

Daniel Karlberg & Daniel Kem, Gothenburg, June 2021

Contents

List of Figures

1

Introduction

The use of distributed systems has grown dramatically in our modern society over the past few decades. Today we see many digital services being distributed such as cloud storage and automotive systems. These distributed services are being used by every sector of our society from banking and finance to more traditional industries such as manufacturing and military. In the past decade we have also noticed a sharp growth in crypto-currencies built on top of distributed systems and distributed computation. Examples of these crypto-currencies are Bitcoin [43] and Etherium [2]. In other words, These systems are part of our daily lives and are vital to the daily operation of our society.

Many of these digital distributed systems form critical points in our society. Faults or failures in these points could lead to catastrophic events if payment systems stop working or if communication systems break down stopping us from communicating across the world. From this perspective it is easy to appreciate the importance of these systems and why it is of utmost importance for us to ensure that they have a high degree of fault-tolerance built into their design. This is a must in order to ensure the availability of the service which the system provides.

A distributed system is a collection of computers or servers, often referred to as *nodes*, which are connected over either a local or a global network of communication links. This system works together, through communication in the form of message-passing, in order to solve a common shared problem or task. Through working together, a distributed system can compute tasks that are too difficult to be solved by single nodes. Distributed systems can also ensure *availability* and *fault-tolerance* of a service due to the fact that even if a smaller set of nodes within the system crashes, the service which the system provides will still be available. If the service provides data storage, then the data can be replicated across the system's node, ensuring fault-tolerance for the data.

In distributed systems, *state-machine replication* is an important category of tasks used in many services. It provides the function of storing data across a set of nodes ensuring both *availability* of the data at any given time, as long as at least one node has not crashed. It also provides *fault-tolerance* through replicating the data over all the participating nodes. Hence, even if some nodes experience faults and lose the data, it can be updated by other non-faulty nodes in the distributed system.

In order for distributed systems to be able to provide a high degree of *availability*

of the service which it provides. The distributed system must be able to deal with faults and failures, both outside the system (communication failures) but also in the system itself (crashes). However, achieving a high degree of *fault-tolerance* in distributed systems is hard due to the asynchronous message-passing nature which distributed systems operate in. It is challenging in these asynchronous systems to detect possible faults and failures of other participating nodes in the distributed system [41, 24].

1.1 Project Purpose

The purpose of this project is to validate, implement and evaluate a distributed system which provides the service of a *Self-Stabilizing Emulation of State-Machine Replication*, researched by Lundström et al [39]. The system is implemented through a stack of algorithms from recent research in the area of *self-stabilization* and *fault-tolerance* [9, 38, 39, 37]. This research allows the system to recover from both common distributed system faults and *arbitrary transient faults*. The algorithmic stack consists of *self-stabilization consensus* and *failure detection* algorithms which together support the implementation of the *self-stabilizing emulation of state-machine replication*. This project aims to implement this research, in order to evaluate if the recent advancements in the area of self-stabilization can be used to improve the *fault-tolerance* of real-world distributed systems. A successful implementation of this system can provide a demonstration that this research has the possibility of making the cloud *self-stabilized* in the long term. In essence, this project will work to answer the following two research questions, (1) Is the recent research in the area of *self-stabilization* and *fault-tolerance* valid and correct? (2) If the research is valid and correct, then what is the performance of our implementation in relation to the systems latency and throughput?

1.2 Fault Model

One of the main attributes of *self-stabilizing* distributed systems is the ability to operate and provide its service in the presence of faults and failures. The implementation that this project provides utilizes asynchronous message-passing systems with no shared memory between computers in the system. The asynchronous distributed system has no universal clock available for the computers in the system. Due to these two factors, the participating computers can only operate on sending and receiving messages with each other.

The fault model which this implementation handles includes crashes of participating computers, communication failures such as packet loss and *arbitrary transient faults*. An *arbitrary transient fault* is a fault that can corrupt communication channels or processors in the system. These transient faults can cause the system to violate the assumptions of which the distributed system was designed to operate in [38].

Research papers which handle *arbitrary transient faults* often limits the effect that these faults can have on the system. One such limitations which is often made is

that, *arbitrary transient faults* cannot corrupt the actual program or algorithm code while the system is running. such a fault could change the design of the distributed system and hence, cause it to not have its original properties [38, 39, 4].

1.3 What is Emulated State-Machine Replication?

A *state-machine* is a computational mathematical model which stores the current state S of a machine. Given some inputs R to the state-machine, it will calculate the *transition* to the next state S' and store the new state S' in the *state-machine*. A real-world abstraction of this model could be, for example, a data-server, which based on new client data inputs need to store the new data and hence, the data-server will *transition* to a new state which incorporates the new data.

State-machine replication is a concept used to increase the *fault-tolerance* of state-machines by replicating the state-machines in a distributed system [33]. Distributed algorithms can be used to ensure that in a fault-free situation, all state-machines of the distributed system will have the most recent state S stored locally.

By storing the current state S in more than one state-machine, also commonly referred to as *replicas*, both the *availability* and the *fault-tolerance* of the service can be increased. The availability can be increased through faster access to a more geographically closer replica and even if one replica is faulty, another one can be accessed for the same data or state. The fault-tolerance can be improved due to the fact that state S is stored at more than one location. Hence, if one replica would crash, the same state S is still available from another replica. If the crashed replica would reboot and start working again within a finite time window, the working replicas in the distributed system could update the crashed and rebooted replica with the current state of the distributed system.

This project implements an *emulated state-machine replication* provided by Lundström et al. [39] which is constructed by the use of distributed *self-stabilizing consensus* algorithms. With emulation, it is meant that the implementation provides a baseline state-machine replication used to demonstrate the developments in the research area of *self-stabilization*. Dolev et al. [11] has previously used emulations of state-machine replication in attempts to demonstrate *self-stabilization* for this type of distributed systems.

1.4 What is Consensus?

The *consensus* problem is a fundamental part of distributed systems and a requirement in order to implement a *self-stabilizing emulation of state-machine replication*. The problem consists of a set of participating nodes in a distributed system that needs to propose and reach an agreement on a shared value v. Lundström et al. [38] define the consensus problem as follows:

(Consensus definition) Every process p_i that is part of the system proposes a value v_i. Let an algorithm solve the consensus. This algorithm has to satisfy the following:

- **Validity -** If v is the consensus, then propose(v) was invoked by some process in the system.
- **Integrity -** A process decides on a value at most once.
- **Agreement -** Once consensus is reached, all processes share the proposed value v.
- **Termination -** All non-faulty processes decide.

This implementation will focus on two types of consensus algorithms for distributed systems, *Binary Consensus* and *Multivalued Consensus*.

The *Binary Consensus* is a subsection of the general consensus problem [38]. A binary consensus solves the task for a set of participating nodes in a distributed system to propose and reach an agreement on a shared binary value v from the set (0,1), i.e. either zero or one.

The *Multivalued Consensus* is an extension of the binary consensus problem which allows a set of participating nodes to also propose and reach an agreement on any shared integer value v instead of just a binary value [39]. It is common for multi-valued consensus algorithms to be constructed from binary consensus algorithms.

Fischer et al. [24] states and explains in their research paper that there exists no solution to the consensus problem in distributed systems which utilizes asynchronous message-passing systems where crashes can go undetected due to the asynchronous nature of these systems. In such asynchronous systems, a failure or fault of just one participating node can cause the entire consensus process to fail. This is why *failure detectors* are necessary in order to solve the consensus problem in distributed asynchronous message-passing systems.

1.5 What is Unreliable Failure Detection?

Asynchronous distributed systems which operate by using message-passing systems are known to be prone to faults connected with the crashing of participating nodes. One of the main problems with asynchronous systems is that it is impossible to know if a participating node in the system has crashed or if the concerned node is just slow on responding to messages [41]. Hence, it is a challenge in these systems to detect when a crash actually has occurred.

This is the reason why *failure detectors* have been researched, in order to solve this problem regarding asynchronous systems. A failure detector commonly runs incorporated on the participating nodes of the distributed system as an independent algorithm. Most failure detectors follow a common design of sending messages to every participating node and monitoring which nodes respond to the sent message. Nodes in the distributed system who are experiencing faults or have crashed will be detected by the other participating nodes in the system. This is due to the fact that these nodes will have a harder time to respond to the failure detector messages of other nodes.

Through this baseline algorithmic approach, crashed nodes in the distributed system can be detected and handled. This will allow other distributed algorithms to be able to continue operating even in situations of crashed participating nodes. An *unreliable failure detector* is a failure detector that will provide the node with a list of participating nodes in the system which it suspects of having crashed [6].

1.6 What is Self-Stabilization?

The research area of *self-stabilization* is part of the general research area of fault-tolerance [7]. Research in the area of fault-tolerance for distributed systems often focuses on the common faults, such as communication faults, packet loss, link failures and crashing of nodes. Meanwhile, the research area of *self-stabilization* focuses on *arbitrary transient faults*. An *arbitrary transient fault* is as mentioned, a fault that can corrupt communication channels or nodes in the system. These transient faults can cause the system to temporarily violate the assumptions according to which the system were designed to operate in [38].

An example of an *arbitrary transient fault* could be a memory corruption which can change the value of a variable in the system, for example a counter variable. This transient fault could cause the system to leave its legitimate state due to an invalid variable value. A distributed system with *self-stabilization* will be able to handle this fault and return the system to a legitimate state within a finite time window [7].

1.7 What is a Global Reset Mechanism?

Arbitrary transient faults can as mentioned change the value of a variable in the system to an arbitrary value. If this transient fault causes a variable to be close to the max integer value of the system (2^{64}), there is a risk of integer overflow. A *global reset mechanism* can be used to handle these situations and prevent integer overflow by resetting variables for the entire distributed system in conjunction [9].

Global reset mechanism is not needed outside the context of *self-stabilization*. Rather, *global reset mechanism* and *self-stabilization* operate together in order to provide fault-tolerant operation of distributed systems. This is mainly due to the fact that a variable counter will realistically never reach the max integer value of 2^{64} in a reasonable time window without a transient fault.

1.8 Related Work

Paxos is a published algorithm by Lamport [34], which provides a distributed system of nodes to reach a consensus over an asynchronous network. The proposed algorithm by Lamport does not make use of any large element from the research area of fault-tolerance and hence lacks properties for more fault-tolerant systems.

The Paxos algorithm is also unbounded which can induce system failures in the case of *arbitrary transient faults* or long run times, depending on the system.

Algorithms in the research area of self-stabilization often build on top of already existing algorithms for distributed systems or distributed computation. In some cases these algorithms have already started the implementation of certain fault-tolerant practices. Research of self-stabilization applies further fault-tolerant practices along with solutions for recovery after the occurrence of the last arbitrary transient faults in order to make the algorithms self-stabilizing.

The failure detector of this project is a further development of Mostéfaoui et al. [41] research of a failure detector which in turn builds on work by Chandra et al. [6] who developed the unreliable failure detector class. The self-stabilizing consensus algorithms proposed by Lundström et al. [38, 39] builds on proposed non-self-stabilizing binary consensus algorithms by Guerraoui et al. [30] and non-self-stabilizing multivalued consensus algorithms by Mostéfaoui et al. [42]. Lundström et al. also expands on research by Zhang et al. [50] who demonstrate a bounded solution for the consensus problem, needed in order to prevent overflow problems of unbounded algorithms.

For the area of state-machine replication, there have been proposed attempts by Dolev et al. [11] to deliver self-stabilizing state-machine replication solutions. However, these solutions do not meet Dijkstra's criteria for self-stabilization [7]. Hence, the algorithms provided by Lundström et al. [39], which are implemented in this project, aims to provide a fully self-stabilizing state-machine replication solution. We note that earlier solutions appeared in [12, 18, 23, 22, 21].

There also exists solutions to coordination of nodes over a network. One example of this is Zookeeper [31], which provides a centralized service for coordination of a system of nodes. However, a centralized solution creates a critical point of failure in the system design and is not optimal for a fault-tolerant focused distributed system.

Other solutions in the area of self-stabilization include the following: Byzantine-tolerant Binary Consensus [27], Consensus for Shared-memory Systems [17], Atomic Snapshot[26, 36], Global Reset [25, 19], Vector Clocks [46], Quorum Reconfiguration [8, 10], Control Plains [3, 28], Reliable Communication [16], Virtual Traffic Lights [5, 40, 44, 47, 48, 20, 21, 13, 14, 15, 49] and Programming Agents [1].

1.9 Our Contribution

We provide, to the best of our knowledge, the first implementation and evaluation of a self-stabilizing emulation of state-machine replication built on self-stabilizing consensus algorithms researched by Lundström et al [38, 39]. Our implementation also incorporates a global reset mechanism proposed by Dolev et al [9].

Our corresponding evaluation of the implementation shows that the implemented distributed system is able to perform its service while also experiencing faults including communication problems, crashing of nodes and arbitrary transient faults

(Chapter 4). The results presented in this thesis show that the system's performance scales decreasingly to the number of nodes in the system and that safety and liveness is guaranteed. However, it is also evident that the local evaluation environment presents challenges in terms of single computer hardware bottlenecks which are clearly seen when a larger number of nodes are introduced into the system, more specifically in our case more than 7 nodes. The evaluation demonstrates that the recent advancements in the research area of self-stabilization and fault-tolerance can contribute to real-world distributed systems with increased levels of overall fault-tolerance, making current systems more robust against faults and failures.

1.10 Document Structure

This first chapter has provided basic theoretical information needed to understand the basics of distributed systems and self-stabilization in order to better understand the rest of the thesis. Chapter 2 presents the algorithm stack implemented by us and the architecture needed for the implemented system to operate correctly. Chapter 3 gives an overview on how the distributed system and its algorithm stack was implemented. Chapter 4 explains the evaluation environment and the evaluation performed on the distributed system to ensure that the implementation is according to the specification, while Chapter 5 presents the results of the evaluation. The thesis will conclude with a conclusion in Chapter 6.

2

The System

This chapter presents the algorithm stack and the including algorithms, as well as the system architecture that provides service of self-stabilizing emulation of state-machine replication.

2.1 The System Architecture

The distributed system which this project implements consists of an architecture containing a stack of distributed algorithms. Every algorithm in the stack provides services for the algorithms in the upper tiers of the stack. The architecture of algorithms function in unison in order to create a distributed system which can provide the service of a *self-stabilizing emulation of state-machine replication* [39]. The architecture is identically replicated on every participating node in the system, meaning that all nodes have a independent local version of the entire stack available and running on the specific nodes host computer. The stack of algorithms operate and progress through communication with the asynchronous message-passing system between the participating nodes in the distributed system. Message-passing over the network which the distributed system operates in, is the only form of communication between the nodes. The algorithmic stack included in the architecture can be found in *Figure 2.1*. Following is a base bottom-up explanation of the algorithmic stack.

Figure 2.1: Architecture Stack of Algorithms

(1) The *User Datagram Protocol* (UDP) [45] is represented at the bottom of the stack, as it is the building block protocol for all asynchronous message-passing in the distributed system (*Section 2.2*).

(2) One step above the UDP in the stack, there is the *self-stabilizing uniform reliable broadcast* [37] which is used to guarantee delivery of specific broadcasted messages in a self-stabilizing manner (*Section 2.3*).

(3) The *self-stabilizing failure detector* [38] is responsible for detecting faulty nodes in and electing a leader from the non-faulty participating nodes of the system (*Section 2.4*).

(4) The *self-stabilizing binary consensus* [38] is used together with the failure detector to reach an agreement on a binary value v amongst the non-faulty participating nodes (*Section 2.5*).

(5) By using x instances of the binary consensus, where x is the number of participating nodes, the *self-stabilizing multivalued consensus* [39] can provide the system with the service of a consensus to reach an agreement on any integer value v (*Section 2.6*).

(6) On the top of the algorithm stack is the *self-stabilizing state-machine replication* [39]. This algorithm uses the multivalued consensus from a layer below to reach an agreement on the next state S' of the distributed system. Once a consensus has been reached, the participating nodes will transition from the current state S to S' agreed to in the consensus (*Section 2.7*).

(7) Connected to all algorithms in the architecture is a *global reset mechanism* [9], which will monitor variables from all algorithms in the stack and initialize a global reset if an integer overflow is imminent (*Section 2.8*).

The system is able to provide the service of a self-stabilizing emulated state-machine replication as long as a quorum of the participating nodes are considered to be non-faulty. Through the self-stabilization of the algorithm stack, the entire distributed system can be fault-tolerant and self-stabilizing.

2.2 User Datagram Protocol

At the basis of almost all distributed systems which operate in asynchronous networks and utilize asynchronous message-passing systems, there is the *User Datagram Protocol* (UDP) [45]. It is through this protocol that all message-passing between participating nodes in the system is done. All algorithms in the stack that the architecture includes utilizes UDP when communicating with other participating nodes in the distributed system. Each node in the system is going to operate from a socket on the computer which the node is running on. It is from this socket which the node is sending and receiving messages (UDPs). Prior to starting the system, every participating node will have knowledge of the IP addresses and sockets of everyone in the system. Everyone will also have an assigned node ID that is known by every participating node in the system. This information allows the nodes to send messages between each other using UDP. The nodes can send messages to specific nodes or broadcast messages to a set of nodes or all nodes. Broadcasting messages utilizing only UDP is referred to as *Unreliable Broadcast*. This broadcast does not spend any

extra resources on ensuring that the message will arrive at the target destination. Hence, *Unreliable Broadcast* does not guarantee delivery of sent messages.

2.3 Self-Stabilizing Uniform Reliable Broadcast

Above the *User Datagram Protocol* in the algorithm stack of the architecture, there is the *Self-Stabilizing Uniform Reliable Broadcast*[1] algorithm by Lundström et al [37]. This uniform reliable broadcast (URB) implementation provides the architecture and the system with the ability to broadcast messages with a guaranteed delivery to every non-faulty participating node in the distributed system, while also being able to handle arbitrary transient faults. Compared to *unreliable broadcasts*, the URB will guarantee that the message will be delivered at the receiving nodes. The algorithms of this architecture utilizes the *self-stabilizing uniform reliable broadcast* for messages which it needs to have guarantees that they will reach other participating nodes. The utilization of *self-stabilizing uniform reliable broadcast* does use more network resources compared to *unreliable broadcasts*. This is due to the fact that *self-stabilizing uniform reliable broadcast* sends more messages to guarantee delivery.

2.4 Self-Stabilizing Failure Detector

As the distributed system operates in an asynchronous network with asynchronous message-passing between the participating nodes, it is difficult to differentiate a non-responding node from a crashed node, it can be the case that the non-responding node is just slow [41]. In our system we utilize a *Self-Stabilizing Failure Detector* proposed by Lundström et al. [38] to solve and circumvent the problems of detecting crashed nodes in asynchronous networks. In addition to detecting crashed nodes in the system, the failure detector will be able to provide the system with an elected leader from the participating nodes. This leader is eventually shared amongst all non-faulty participating nodes.

The self-stabilizing failure detector operates by using unreliable broadcasts when communicating over the network. Hence, no resources are spent on guaranteeing delivery of sent broadcasts. Every participating node is going to have an individual failure detector running locally. The detector will include its local information about suspicious nodes in the messages sent. This ensures that local knowledge is shared with the non-faulty nodes in the system. Through this process the information of the detectors is propagated in the system, which allows the participating nodes to view the system in the same way.

The failure detector (see *Algorithm 1*) achieves its task with one main variable over continuous communication rounds. The variable is an array named $count_i[]$. The array is used to keep a count on the number of times a node has been suspected. Every index of the $count_i[]$ array represents a participating node, a node's position is equal to its own node ID - 1. The $count_i[]$ array can be used to elect a leader for the

[1]Implementation provided by O. Lundström and C. Kou (Chalmers, 2020) [35, 37].

system. This is achieved by electing the node with the least number of suspicions in the array, in other words, the node with the best performance.

The detector completes three steps for each communication round:

1. In the first step, the local node will continuously broadcast an ALIVE message containing $count_i[]$ and wait for a quorum of RESPONSE messages from other nodes to the broadcasted ALIVE message. The RESPONSE message contains the responding nodes $count_j[]$ array and a list of responders to p_j previous communication rounds ALIVE broadcast.

2. The second step consists of merging the list of responders to the local nodes ALIVE message with the ones received in the RESPONSE messages of the same communication round. If a node p_k is not in this merged list, then this node has failed to respond to an ALIVE message. This is a sign that the node might be experiencing faults, hence p_i will increment $count_i[p_k]$ which will increase the suspicion level of the node p_k.

3. In the final step and third step before starting the next communication round, the local node p_i will save the responders of this rounds broadcasted ALIVE message. This list is sent in future RESPONSE messages for the received ALIVE broadcasts.

When the nodes receive ALIVE from other nodes in the system, the receiving node will merge the local array $count_i[]$ with the received $count_j[]$. This is done through taking the maximum for each position in the two arrays. This process allows the information of suspicious nodes to be shared among all non-faulty nodes in the system.

The self-stabilizing failure detector also includes a δ value. This value is specific to the system specifications, which the detector is coupled together with. The decided δ value is used together with a $check()$ function which inspects the $count_i[]$ array for arbitrary transient faults. If a gap between two values in the array is larger than δ, there is a possibility that a transient fault has occurred. If this is the case, the $check()$ function will restore the $count_i[]$ array to a non-transient fault state.

Algorithm 1: Pseudo-Code for Self-Stabilizing Failure Detector for p_i. For a full in-depth and detailed algorithm, see reference [38].

```
1  Local constants, variables and their initialization:
2
3  macro check(); /* checks the count array for violations of maximum allowed
     gap (δ) */
4  operation leader(); /* select the node with least number of suspicions for
     leader */
5  recFrom := P; /* a list of the nodes who responded to the most recent ALIVE
     round */
6  count[0..n − 1] := [0..0]; /* the array for the number of suspicions on each node
     */
7
8  do forever begin
9      Broadcast ALIVE();
10     Wait for RESPONSE() from quorum;
11     foreach node p_k that did not respond:
12         Increment count[k] by one;
13     Store the nodes who responded in recFrom;
14     Check for δ violations in count[] with check();
15 upon ALIVE() arrival from p_j begin
16     Merge the local count with the one received;
17     Check for δ violations in count[] with check();
18     Answer p_j by sending RESPONSE() back;
19 upon RESPONSE() arrival from p_j begin
20     Merge the local count with the one received;
21     Check for δ violations in count[] with check();
```

2.5 Self-Stabilizing Zero-Degrading Indulgent Binary Consensus

The *Self-Stabilizing Zero-Degrading Indulgent Binary Consensus* proposed by Lundström et al. [38] is a fundamental part of the implemented distributed system. The algorithm provides a binary consensus service which is both *zero-degrading* and *indulgent*. By zero-degrading it is meant that a failure of one consensus round has no impact on future runs of the algorithm. The indulgent property of the algorithms means that an arbitrary behavior of the failure detector never violates the safety requirements, even if it compromises the system's liveness [29]. The algorithm will provide the system with a binary consensus for a value $v \in [0, 1]$.

Communication in the algorithm is done by using both unreliable broadcasts and self-stabilizing uniform reliable broadcasts (URB). Uniform reliable broadcasts are used to broadcast the result of the consensus, this is done in order to ensure that every non-faulty participating node gets the consensus results.

The implementation of the consensus algorithm utilizes five main variables in its operation and two phases. (1) r_i is the current local round number for the current run of the consensus. (2) the estimate array, $est[0..2]$ contains three variables. $est[0]$ is the value proposed to the consensus by the local node. $est[1]$ is the consensus value reached in the first phase. $est[2]$ is the consensus value reached in the second phase and also the result of the binary consensus. (3) $myLeader$ is the current elected leader of the system, supplied by the self-stabilizing failure detector. (4) $newR$ is the aggregated round number, if this is larger than r_i, then the new r_i will be $newR$. The aggregated round number is received from other nodes in the systems through broadcasts from other non-faulty nodes. This ensures that a slow node will get the aggregated round number from other nodes when receiving messages. (5) The final variable is $txDes$, which is the uniform reliable broadcast transmission descriptor. It will allow us to check if an URB transmission has finished or if it is still being dealt with by other nodes.

The operation of the self-stabilizing binary consensus (see *Algorithm 2*) consists of two phases (0 and 1). When initializing a run of the binary consensus, $est[]$ is set to $[v,\perp,\perp]$ before starting the consensus, where v is the proposed value by the local node. Before entering the first phase of the algorithm, it will check if there already exists a result on $est[2]$. If that is the case, then the algorithm will URB the consensus results if no other URB is active. The algorithm will also fetch the current elected *leader* from the failure detector and update the current consensus round number.

1. In *phase 0*, the node will broadcast $est[0]$ using unreliable broadcast, until $n - t$ *phase 0* messages have been received with matching round number r, from unique nodes. One of the unique nodes needs to be the elected leader. If a majority of the messages received has the same elected leader as the local node, then $est[1] \leftarrow v$, where v is the received value in the broadcast from the leader. If a majority did not have the same elected leader, then $est[1] \leftarrow \perp$.

2. In *phase 1*, the node will broadcast $est[1]$ using unreliable broadcast, until $n - t$ *phase 1* messages have been received with matching round number r, from unique nodes. If all *phase 1* messages contain the same $est[1]$ value, the node will decide on the value in $est[1]$ and use URB to broadcast the consensus result. If the responses contains both a value and/or $\perp$, then the consensus will restart *phase 0* with $est[0]$ set to the value in $est[1]$.

When receiving a phase broadcast from a node p_j and this broadcast contains an $est[1]$ value. The node will check if the local $est[1] = \perp$. If that is the case, the node can update the local $est[1]$ with the value from the broadcast. If the received phase message was a broadcast, a response is also sent to the broadcasting node.

Once a node receives a URB containing a consensus result v, the node will check if there already is a local result in $est[2]$. If not, then the node will put the value v from the URB in $est[2]$, $est[2] \leftarrow v$. This process allows the set of participating nodes to reach a binary consensus as long as there is a quorum of non-faulty nodes.

Algorithm 2: Pseudo-Code for Self-stabilizing Indulgent Zero-degrading Binary Consensus for p_i. For full in-depth and detailed algorithm, see reference [38].

```
1  Local constants, variables and their initialization:
2
3  est[] = [⊥,⊥,⊥];
4  myLeader = ⊥;
5  operation: propose(v) := est[v,⊥,⊥];
6
7  while true do
8      If est[2] ≠ ⊥ ∧ txDes = false:
9          urbBroadcast(DECIDE(est[2]));
10         continue;
11     Get leader ID from failure detector;
12     r = max(r, newR);
13     repeat broadcast PHASE 0(r, est[0], leader) until: /* Phase 0 */
14         n − t (majority) received phase 0 messages with matching r;
15         received phase 0 broadcast from leader with matching r;
16     If majority of phase 0 messages had the same leader as local node:
17         est[1] ← est[0] received from leader;
18     repeat broadcast PHASE 1(r, est[1], leader) until: /* Phase 1 */
19         n − t (majority) received phase 1 messages with matching r;
20     Let rec = [est[1] values received in phase 1];
21     If rec = [v] → urbBroadcast(DECIDE(v));
22     If rec = [v,⊥] → est[0] = v;
23     If rec = [⊥] → continue;
24 upon PHASE n(r, est[1], leader) from p_j begin
25     if Phase 1 ∧ est[1] = ⊥ then est[1] ← v (from phase message);
26     if Phase n was a broadcast then respond Phase n to p_j;
27 upon DECIDE(v) from p_j begin
28     if est[2] = ⊥:
29         est[2] ← v;
```

2.6 Self-Stabilizing Multivalued Consensus

For the *Self-Stabilizing Emulation of State-Machine Replication* to be able to create a consensus among the set of nodes, on the transition from the current state S to the next state S', the set of nodes need a multivalued consensus. A multivalued consensus allows for the distributed system to reach an agreement on any integer value v, compared to a binary value.

The *Self-Stabilizing Multivalued Consensus* provided by Lundström et al. [39] continues on the implementation of and utilizes the self-stabilizing binary consensus as a building block, in order to provide a solution to the multivalued consensus problem. The proposed algorithm uses x instances (or objects) of the aforementioned

self-stabilizing binary consensus solution, where x is the number of nodes in the distributed system. Every instance of the binary consensus has its own set of variables, explained in *section 2.5*. The instances can also be identified through a sequence number in $[0, \ldots, x-1]$.

The implementation of the algorithm utilizes three main variables in its operation to achieve a multivalued consensus. (1) The $proposals[]$ array keeps the integer value proposals made by every node p_k in the distributed system. Hence, every node has a unique position in the array (p_k position is $proposals[p_k]$). (2) The $BC[]$ array stores all the x instances of the binary consensus for the specific system. (3) The final variable is $txDes$, which is the uniform reliable broadcast transmission descriptor. It will allow us to check if an URB transmission has finished or if it is still being dealt with by other nodes.

The algorithm (see *Algorithm 3*) operates by proposing an integer value. This is achieved through self-stabilizing uniform reliable broadcasting of the nodes proposed integer value v, let us refer to this broadcast as a $proposalURB(v)$. The usage of URB ensures the delivery of the sent proposal from the sending node. When a node p_i in the system receives a $proposalURB(v)$ from p_k in the system, p_i will store the proposal in $proposals_i[p_k]$. If the first self-stabilizing binary consensus instance ($BC[0]$) is not already activated and running, the node will start the consensus process by activating the first binary consensus instance. The algorithm will continuously activate binary consensus instances iteratively until an instance of the binary consensus returns the results $v = 1$. The same index in $BC[i]$ which returned the result $v = 1$, is going to be used to select the reached consensus value from the $proposals[i]$. If $proposals[i] = \perp$ or if all of the instances of $BC[]$ return the result $v = 0$, then the consensus has failed and needs to be restarted. The proposal array can be $\perp$ if the issued node did not propose any integer value for the consensus, or if network faults caused the consensus to finish before the proposal had arrived at every node in the distributed system.

Algorithm 3: Pseudo-Code for Self-Stabilizing Multivalued Consensus for p_i. For full in-depth and detailed algorithm, see reference [39].

```
1  Local constants, variables and their initialization:
2
3  v = ⊥; /* local proposal value */
4  proposals[] = ⊥-s; /* array of arriving proposals */
5  BC[]; /* array of x binary consensus objects */
6  txDes = ⊥; /* URB transmission descriptor for decision sharing */
7
8  do forever begin
9      If v ≠ ⊥ ∧ txDes = false:
10         urbBroadcast(PROPOSAL(v));
11     Wait for BC[i] to return result; /* i = 0 */
12     If BC[i] = 0:
13         activate BC[i + 1];
14     If BC[i] = 1:
15         return proposals[i];
16         consensus finished;
17     If i = NumberOfNodes:
18         consensus failure;
19         restart consensus;
20 upon PROPOSAL(v) from p_j begin
21     let proposals[p_j] = v:
22     If BC[0] ≠ active:
23         activate binary consensus instance BC[0];
```

2.7 Emulated State-Machine Replication using Total-Order Broadcast

The underlying stack of self-stabilizing distributed algorithms is used to implement a *Self-Stabilizing Emulation of State-Machine Replication* proposed by Lundström et al. [39]. In its essential form, the emulated state-machine replication will continuously use the self-stabilizing multivalued consensus to reach an agreement on the transition from the current state S of the distributed system to the next state S'. The multivalued consensus will in return use the self-stabilizing failure detector and the self-stabilizing binary consensus in order to achieve its task of a consensus on an integer value. From this perspective, it can be seen how the algorithm stack together provides the functionalities required to implement the *self-stabilizing emulation of state-machine replication.*

The emulation of state-machine replication uses total-order broadcast based on the self-stabilizing uniform reliable broadcast in order to send ordered messages with unique sequence numbers. In order not to flood the networks communication channel or the nodes with self-stabilizing uniform reliable broadcasts which use more network

resources, total-order broadcasts are only sent once per a given time window. The time window is based on an estimation of the round trip time, ensuring that a broadcasted total-order message has a chance of reaching the target before a new uniform reliable broadcast is made. When a node p_j receives total-order messages, the node will put the them in a sorted and ordered bag of undelivered messages. The algorithm utilizes a *synchronization phase*, where every non-faulty participating node in the system will broadcast the maximum consecutive undelivered sequence number received. Through this process, the non-faulty nodes will be able to discover the maximum consecutive undelivered sequence number shared amongst the nodes in the system. This sequence number can be proposed to the multivalued consensus and if a consensus can be reached, then the participating nodes can deliver the messages up to and including the sequence number from the bag of undelivered messages to the state-machine. This process ensures that the new state S', which includes the recent delivered messages is replicated and shared between the non-faulty nodes in the distributed system. In other words, a state-machine replication.

The algorithm (see *Algorithm 4*) for self-stabilizing emulation of state-machine replication operates by using three instances (or objects) of the self-stabilizing multivalued consensus. Each of these objects have in turn x binary consensus objects available (where x is the number of nodes in the system). Three instances of the multivalued consensus is used to have the algorithm be bounded in the number of available instances. This is to prevent overflow problems from a system operating over longer run-times, using an unbounded number of multivalued instances. Hence, at any given time, a maximum of two multivalued consensus instances are active. The instances are recycled, garbage collected and reset for re-use. This makes the algorithm bounded with the number of multivalued instances used.

As aforementioned, the algorithm utilizes a *synchronization phase* to find a common denominator with the maximum consecutive undelivered sequence number shared amongst the nodes in the system. This phase also provides the participating nodes with the information of which the latest completed multivalued consensus instance each node has accessed. This allows the nodes to know when everyone in the system is on the same instance, which also allows for a new instance to be initialized (with $proposalURB(v)$, see *Section 2.6*) and the old used instance to be recycled, garbage collected and reset for re-use. The synchronization phase also allows for nodes that have experienced faults but recovered to gain information and update itself with the current state of the state-machine and the current activated multivalued consensus instances. In case of faults which cause wrongful activation of the instances, a trigger exists which resets all three multivalued consensus instances.

Algorithm 4: Pseudo-Code for Emulated State-Machine Replication using Total-Order Broadcast for p_i. For full in-depth and detailed algorithm, see reference [39].

```
1  Local constants, variables and their initialization:
2
3  currentInstance; /* last sequence of the instance result read */
4  allInstance[]; /* currentInstance of nodes in the system */
5  maxReady; /* local maximum consecutive undelivered sequence number */
6  allMaxReady[]; /* maximum consecutive undelivered sequence number of
     nodes in system */
7
8  while true do
9      repeat broadcast SYNC until:
10         SYNCack received from all non-faulty nodes with matching r;
11     let v = min(allMaxReady[]);
12     If foreach non-faulty p_k;
13         allInstance[p_k] = currentInstance ∨ (currentInstance + 1) mod 3 ;
14         Then urbBroadcast(PROPOSAL(v));
15 upon SYNC(r) from p_j begin
16     Send SYNCack(maxReady, currentInstance, r) to p_j;
17 upon SYNCack(maxReady, currentInstance, r) from p_j begin
18     if matching r;
19         allMaxReady[p_j] = maxReady;
20         allInstance[p_j] = currentInstance;
```

2.8 Global Reset Mechanism

The stack explained in the sections above can together provide a distributed system for *Self-Stabilizing Emulation of State-Machine Replication*. However, for the system to be completely self-stabilizing, there is a need for a global reset mechanism. This is due to the fact that the listed algorithms in the stack utilizes unbounded variables which are not reset together with the instances, for example counters. It is not viable to suppose that these counters would reach integer overflow limits (2^{64}) from normal operation, it would take too much time. An arbitrary transient fault could however cause variables to jump to a value close to the integer overflow and risk system malfunction if it were to reach it. Hence, a global reset mechanism is not needed outside the context of self-stabilization. However, it is a requirement for a fully self-stabilizing system.

An algorithm (see *Algorithm 5*) for *Global Reset Mechanism* is provided by Dolet et al. [9], which is a self-stabilizing state-transition for global reset mechanism. The algorithm operates by continuously monitoring unbounded variables. If a variable is at risk of integer overflow, a global reset mechanism will be invoked. The mechanism will transition every non-faulty participating node in the system through a series of

six states in a synchronized manner. This allows the system to reset the variables to a safe configuration in the final state of the synchronized mechanism, in conjunction with each other, ensuring that everyone resets the variables at the same time. The synchronized transition is carried out through a process where the participating nodes *echo* their current state in the reset mechanism. This allows the nodes to know the mechanism state of every non-faulty node in the system. Once everyone is in the same state, they can transition to the next state in the mechanism together. In the case of a fault, where a participating node falls out of order in the mechanism state transition, meaning that a node is two states behind or two states in front of any other node in the system. Then the entire reset mechanism will restart for every non-faulty node.

Algorithm 5: Pseudo-Code for Global Reset Mechanism for p_i. For full in-depth and detailed algorithm, see reference [9].

```
1  Local constants, variables and their initialization:
2
3  myCurrentState = 0; /* my current state */
4  echoStates[]; /* current state of other nodes in the system */
5  macro transition(myCurrentState); /* 1 → 2 | 2 → 3 | 3 → 4 | 5 → 0 */
6
7  while true do
8      If reset needed ∧ foreach non-faulty p_k;
9          echoStates[p_k] = 0;
10         Then myCurrentState = 1;
11     If foreach non-faulty p_k;
12         echoStates[p_k] = myCurrentState ∨ (myCurrentState + 1) mod 5;
13         Then myCurrentState = transition(myCurrentState);
14     If ∃ p_k ∧ non-faulty;
15         echoStates[p_k] = 1 ∧ myCurrentState = 0;
16         Then myCurrentState = 1;
17     If myCurrentState == 5;
18         reset unbounded variables;
19     Broadcast ECHO(myCurrentState);
20 upon ECHO(myCurrentState) from p_j begin
21     echoStates[p_j] = myCurrentState;
```

3

Implementation of the System

This chapter presents how the system and architecture is implemented in the programming language RUST and how it is able to provide the service of self-stabilizing emulation of state-machine replication.

3.1 The Programming Language RUST

This project was implemented in the programming language RUST [32]. RUST is a statically typed language which was used for the implementation of the self-stabilizing uniform reliable broadcast, due to the fact that RUST was developed for performance focused systems. This thesis project continues the utilization of RUST due to the legacy code and a set of good characteristics such as safe concurrency operations between threads and no run-time or garbage collection for improved performance. RUST also has a rich type system which includes a ownership model with both memory and thread-safety. These characteristics in combination with the legacy infrastructure allows RUST to be a suitable programming language for this thesis project which aims to increase fault-tolerance of distributed systems.

3.2 Threads

The implementation which provides the service of a self-stabilizing emulation of state-machine replication is constructed by using two threads. One thread is used for running the *algorithms* while the other one is responsible for handling the *communication*. The two threads will in conjunction provide the service of a self-stabilizing emulated state-machine replication. The implementation is divided into two threads for efficiency. The algorithms send a large quantity of messages and hence, also receive a large quantity. This is occurring while the algorithms also have to complete computations. For this reason, the implementation is divided between two threads, the *algorithm* thread and the *communication* thread.

3.2.1 Algorithm

The algorithm thread operates the majority of algorithms in the architecture stack. Specifically, the thread handles the computation of the algorithms from the failure detector to the state-machine replication in the stack. When an algorithm in the

stack, which is controlled by the thread, needs to send or broadcast a message, the algorithm calls a function with the data it wants included in the message, what type of message it is (e.g. *alive* or *sync*) and if the message should be broadcast or direct message to a specific node. This information is transferred to the communication thread which sends the message through the message-passing system. For messages that need to be broadcast using self-stabilizing URB, a separate function is called.

The algorithm thread also continuously asks the communication thread to fetch new messages from its buffer and hand over the data and original sender to the algorithm thread. Through this method, the thread can receive new messages and complete computations on the data and progress in the algorithm.

3.2.2 Communication

The thread responsible for communication handles all message-passing, which includes sending and receiving UDPs at the socket level. It continuously listens to the socket for incoming UDPs and stores them in respective buffers depending on the type of message (e.g. *alive* or *sync*). When the algorithm thread requests a new *alive* message, the communication thread can fetch a message from the correct buffer and transfer it to the algorithm thread.

The communication thread also handles the self-stabilizing uniform reliable broadcast algorithm [37]. The algorithm thread sends uniform reliable broadcasts through a function call, which in turn tells the communication thread to send an URB using the self-stabilizing algorithm. The communication thread runs the algorithm continuously in order to ensure delivery of messages sent as URBs. Figure 3.1 shows an example of operations between the two threads.

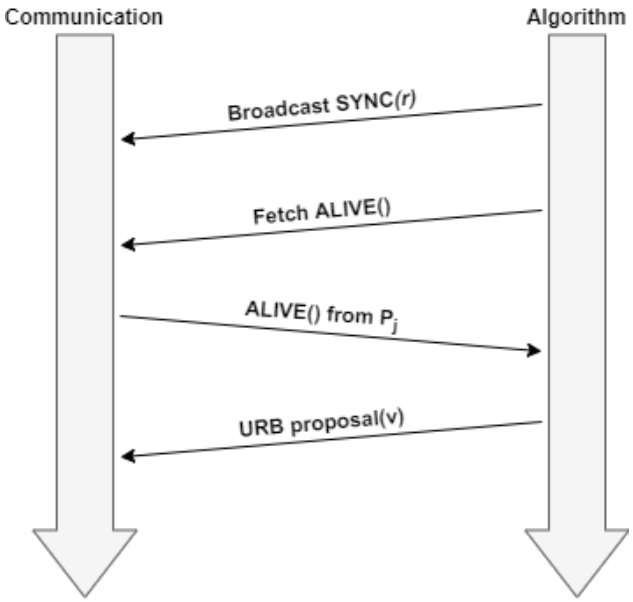

Figure 3.1: Example of operations between the Communication and Algorithm threads.

4

Evaluation

The evaluation chapter is divided into two main parts. Firstly, the evaluation environment is described and its use explained. Secondly, the experiment plan describes the details of the experiments, the manner in which the experiments will be carried out and what data we will gather.

4.1 Evaluation Environment

The evaluation of the implementation was completed in a local evaluation environment, using a local machine. The local environment allows us to deploy our application for easy verification of our implementation of the research. Through this environment we can perform both validation of the research and evaluate the performance of our implementation.

The environment operates by assigning an individual socket for each participating node part of the current evaluation configuration on the local machine. A unique ID is also assigned to each node operating on a socket address. The nodes communicate with each other over the loop-back interface address (127.0.0.1) by using the message-passing system. The participating nodes have information about each other's ID and socket addresses from the initialisation of the evaluation configuration.

Through this environment, we are able to validate if the research and implementation is performing according to the specifications of the research papers. The environment also allows us to log data from the evaluations runs for inspection and analysis of the implementation's performance. In order to evaluate self-stabilization and fault-tolerance, faults can be manually inserted through executing a piece of code while the implementation is running within the evaluation environment. This allows us to evaluate how the implementation handles faults and if they are handled in accordance with the specifications.

4.2 Experiment Plan

When evaluating the system there are many different types of experiments which can be carried out to test both the functionality of the algorithms as well as their performance. For this reason an experiment plan is utilized. The experiment plan of

our implementation will provide information needed to answer the research questions of this thesis:

- Is the recent research in the area of *self-stabilization* and *fault-tolerance* valid and correct?
- If the research is valid and correct, then what is the performance of our implementation in relation to the systems latency and throughput?

Our evaluation criteria consider the ability to handle the faults (*fault-tolerance*) explained in the fault model and the time for algorithm operations to complete (*latency*). Due to the fact that the distributed system continuously makes invocations for new operations from algorithms in the stack, we can provide information about the systems *throughput* i.e. the number of operations completed over a period of time, by inverting the data gathered from the latency experiments. We also consider the scalability of the system, i.e. how the performance of our implementation is affected by the increase of participating nodes in our experiments.

Evaluation and experiments will be executed in the local evaluation environment. For the experiments, 5 different configurations of participating nodes will be used. The five configurations consist of 3, 5, 7, 9 and 11 participating nodes. The reason for these configurations is due to the fact that the quorum of the system is calculated with $t = n/2$, where n is the number of nodes and t is the number of nodes that are allowed to be faulty. The quorum is used in the algorithms for finding and/or waiting for a set of nodes. At the different configuration, the number of nodes needed for this set changes. This is how the number of nodes in the configurations was selected.

4.3 Experiment description

This section provides a detailed list of experiments part of the experiment plan.

4.3.1 Latency

The list for latency related experiments includes the following:

1. **Latency of the self-stabilizing failure detector.** The binary consensus algorithm depends on having a reliable leader in the system in order to function properly. This is why it is important for the system to have a failure detector which is reliable and has a low latency in order to facilitate decision making. This is due to the fact that if different nodes follow different leaders, the binary consensus algorithm will not come to an agreement on a shared value.

   The performance of the failure detector is only impacted by a slow node if that node is needed in the quorum for the majority. A slow node can be described as a node with a lower ability to respond, this can be due to message transmission time or CPU constraints. This means that the performance of the failure detector is expected to be high and therefore the latency to be

low even if more nodes are introduced. When running the experiment locally we expect that the biggest factor for its performance is the hardware which eventually will become a bottleneck and it will result in a significant jump in latency when this happens.

We define latency for the failure detector as the time from the event of the current leader failure to the time a new leader has been observed by all working nodes. The experiment was repeated for a total of 10 times, with best and worst cases removed in order to calculate an average latency for each configuration.

2. **Latency of the self-stabilizing binary consensus.** The latency of the self-stabilizing binary consensus is the pillar for the self-stabilizing multivalued consensus and therefore also for the self-stabilizing emulated state-machine replication implementation. Because of this we expect the latency of the binary consensus to be lower than the latency for multivalued consensus. Furthermore, we expect the latency to increase significantly since there is a need to coordinate with all of the working nodes and therefore the amount of messages needed will grow significantly based on how many nodes are in the system.

   The latency of the binary consensus is defined as the time from when a proposal to the consensus was made, until a consensus was reached from the perspective of each node. An average latency was calculated for each configuration by taking the latency of 100 consecutive consensus agreements for each configuration of nodes.

3. **Latency of the self-stabilizing multivalued consensus.** The latency of the self-stabilizing multivalued consensus is an important metric since this algorithm is continuously used by the emulated state-machine replication for reaching agreement on the next state transition. The latency experiment will provide an indication on the system's ability to continuously reach a multivalued consensus agreement. Similarly to the binary consensus algorithm we expect the latency to grow significantly based on the number of nodes in the system, due to the fact that the quorum size will increase as an effect of increased message cost with more participating nodes.

   Latency is defined as the time from the event of a multivalued proposal until a consensus was reached from the perspective of each node. Similar to the binary consensus experiment, an average latency was calculated for each configuration by taking the latency of 100 consecutive consensus agreements for each configuration of nodes.

4. **Latency of the global reset mechanism.** We measure the latency of the global reset mechanism in order to understand how long it takes for the distributed system to reset a variable and how this time scales with the increase of participating nodes. The latency of this part of the stack is important, due to the fact that the system cannot progress while a global reset is occurring.

   In terms for the global reset mechanism, the latency is defined as, the time

from the event of a node issuing a global reset until the last node has reset the variable to a safe state, i.e. the time for the entire system to reset the variable. An average latency was calculated by gathering the latency of the global reset mechanism 10 times for each configuration, with best and worst case removed. This provides us with an indication of the average time for our implementation to reset the distributed system with an increasing number of participating nodes.

4.3.2 Throughput

The list for throughput related experiments includes the following:

5. **Throughput of the self-stabilizing state-machine replication.** The self-stabilizing emulation of state-machine replication utilizes the entire algorithm stack in order to provide its state-machine service. It is therefore of interest to gather information on the throughput of the service, in order to demonstrate how many state transition operations the system can complete under a given time period.

   We define the throughput as the number of total-order deliveries completed over a predefined period of time. We run the entire system and log the number of deliveries completed over 400 seconds, with timestamps for each delivery. This allows us to calculate the average time for a delivery to be made and investigate the change in number of deliveries per second over the predefined time period.

4.3.3 Self-stabilization and fault-tolerance

The list for self-stabilization and fault-tolerance related experiments includes the following:

6. **Node crash failures.** To evaluate the system's ability to handle crashed nodes, we selectively suspend the operation of participating nodes. Through this experiment we iteratively suspend the operations of one node at a time, until we reach the threshold t, where the crashed node is required in the quorum.

   According to the specifications of the algorithms in the stack, the system should be able operate while $t < n/2$. Hence, once $t = n/2$ the state-machine replication is expected not to be able to reach agreements on the next state of the distributed system. In conjunction with the experiment, the failure detector should also be able to classify the crashed nodes as faulty.

7. **Leader failure.** To evaluate the system's ability to handle the crashing of the currently elected leader, we suspend the operation of the current leader. Through this process we can monitor the process of the failure detector to detect that the leader has stopped operating and that a new leader is elected. This allows us to monitor the liveness of the system, i.e. termination.

The failure detector in the system stack should be able to detect the suspended leader and propagate this information throughout the system. A new leader should automatically be elected and shared amongst all non-faulty participating nodes. The system's ability to conduct this process is known as the safety property of the algorithm and the performance of it is the liveness property which is described in the experiment covering the latency of the failure detector.

8. **Demonstrating recovery from synthetic memory corruption.** Due to the fact that arbitrary transient faults cannot be simulated, we instead demonstrate the system's ability to recover from synthetic memory corruption. This process is accomplished by including components for fault injection into the program code. The faults corrupt key variables such as sequence numbers and the goal is to demonstrate that the system can handle this. This allows us to demonstrate that the system is able to recover from the synthetic memory corruption and return to a legal state of execution.

   We demonstrate the system's ability to handle these faults by the injection of synthetic memory corruption faults into objects of the binary consensus and the failure detector.

   We inject components for fault injection into the program code of the binary consensus, which modify the current round number of the activated binary consensus instance. By design of the algorithm, the modified round number will be shared amongst all working nodes if the modified round number is larger than the current round number of the other nodes. If the modified round number is lower than the current round number of the other nodes, the node with the injected fault will just copy and use the round number of a working node.

   With the failure detector, the injection is done to the $count_i[]$ array of a participating node p_i. Through the process of including components for fault injection into the program code, we can increase some position of the array to a large value, which should trigger the system's arbitrary transient fault handling. The $check()$ function should capture this fault and return the array to a legal state for continued operation. Note that other variables which are not directly protected from arbitrary transient faults are handled by the global reset mechanism [9], which resets the variable to a safe state in the case of an arbitrary transient fault.

5

Results

This chapter presents the results and discussion about the experiments performed on the system in order to evaluate the throughput, latency and fault-tolerance of the implementation.

5.1 Experiments results

The following is the results of the experiments described in section 4.3. The experiments demonstrate the functionality of the system in terms of fault-tolerance and self-stabilization, while also providing an indication of the performance of the implementation.

5.1.1 Experiment 1: Latency of the self-stabilizing failure detector

We measure the latency of the failure detector because algorithms in the stack depend on having a valid and reliable elected leader in the system, shared by the working nodes. A failure detector with a high latency could cause problems or delays in the facilitation of the decision making.

The experiment is anticipated to show a small increase in the latency, this is due to the fact that the failure detector only requires a quorum of the fastest nodes in the system in order to progress. Hence, if a slow node is added to the system, which is not required in the quorum, the latency should not increase by a large margin.

The results of experiment 1 which is shown in figure 5.1 shows the latency of the self-stabilizing failure detector. We note that the performance is relatively unaffected by the number of nodes present in the system, this is expected since the failure detector only needs to wait for the majority quorum of the fastest nodes. However, we also note that when 11 nodes are present in the system there is a jump in latency which can be the case since the results are derived from this algorithm running alongside the other parts of the system as explained in the system stack. Because of this the hardware seems to become the bottleneck when running many nodes.

In comparison to the anticipated results, the system's failure detector latency is in-line with what was expected from the results beforehand.

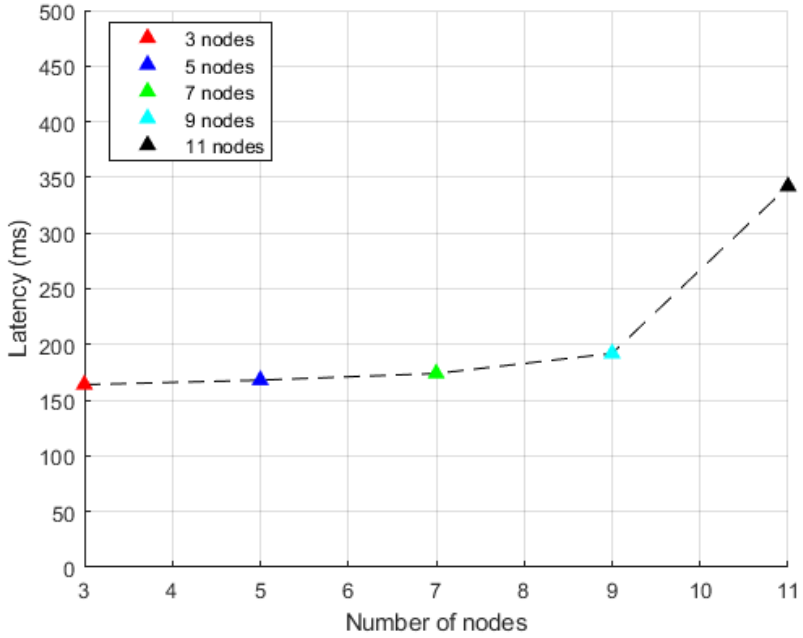

Figure 5.1: The result of experiment 1, showing the latency of the self-stabilizing failure detector.

5.1.2 Experiment 2: Latency of the self-stabilizing binary consensus

The self-stabilizing binary consensus sits at the core of the stack and is a vital component in the system's ability to facilitate decisions. It is directly used in conjunction with the upper layers of the stack, hence latencies in the binary consensus will affect the total system. It is therefore of importance to measure the latency of the binary consensus to find out how the time to reach consensus and decisions scales with the number of participating nodes in the system.

We expect the latency of the binary consensus to notably increase with the number of participating nodes. This is mainly because of an increase in the message cost in order to coordinate all the working nodes through the algorithm's phases in order to reach a valid consensus.

The results of experiment 2 is shown in figure 5.2 which shows that the latency of the self-stabilizing binary consensus grows significantly with the number of participating nodes. However, it is hard to understand how much of the latency increase is due to hardware limitations of the testing environment. Therefore the latency growth in reality may not be as dramatic as shown in the figure.

When compared to the expected results, we find that the results provided by the experiment closely conforms with our anticipated expectations.

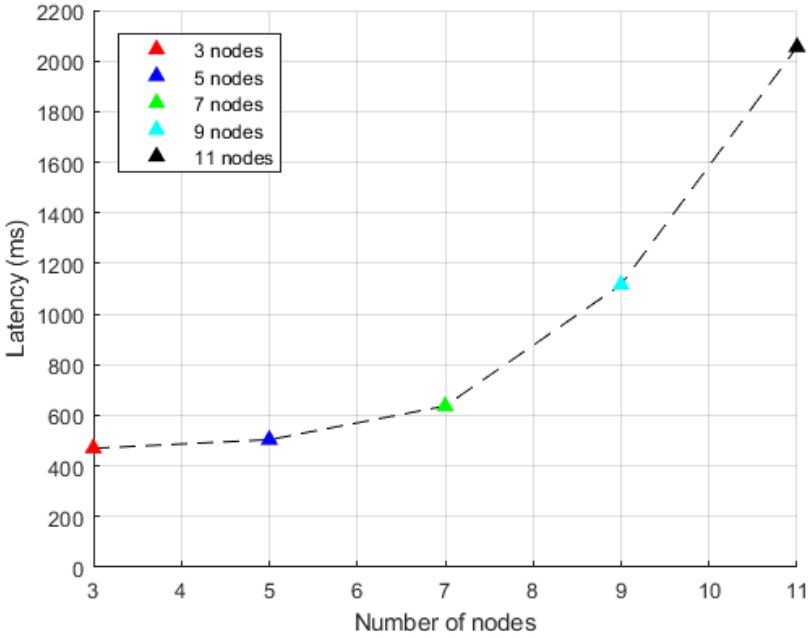

Figure 5.2: The result of experiment 2, showing the latency of the self-stabilizing binary consensus.

5.1.3 Experiment 3: Latency of the self-stabilizing multi-valued consensus

Because the application is directly linked to the multivalued consensus algorithm it is important to evaluate its performance, this is done by measuring the latency.

We expect the latency of the multivalued consensus to be higher than the binary consensus latency, this is because the multivalued consensus is dependent on the binary consensus algorithm and its performance. However, as in the case for the latency of binary consensus we also expect the latency of the multivalued consensus to grow significantly when more nodes are introduced into the system.

The results for the latency of the self-stabilizing multivalued consensus is presented in the graph shown in figure 5.3. The first observation that is made is that the latency for the setup of 11 nodes is considerably higher compared to the other setups, this is likely because the hardware becomes the bottleneck in this system since it is not made to be run on a single machine. The results for this comes from the average of the first 100 consensuses which also explains why the difference is mostly noticeable in the setup for 9 and 11 nodes, this is also something which impacts the throughput of the emulated state-machine replication which we present in figure 5.4.

When we compare the results with our expectations they confirm what we were expecting. However, the hardware bottleneck seems to be a bit excessive specifically for 11 nodes. This is also confirmed when checking the results for the throughput of the self-stabilizing state-machine replication presented in experiment 5.

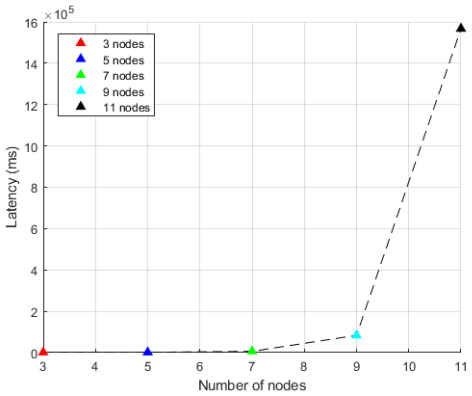

Figure 5.3: The result of experiment 3 showing the latency of the self-stabilizing multivalued consensus.

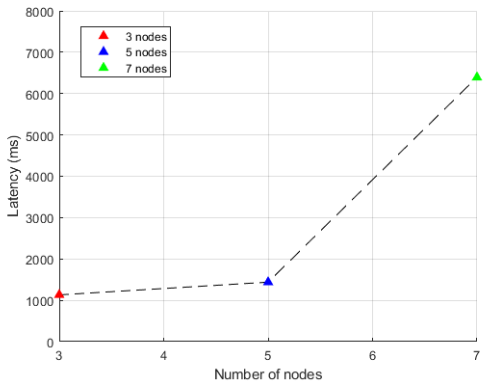

Figure 5.4: A zoomed version of figure 5.3 showing latency for configurations with 3 to 7 nodes.

5.1.4 Experiment 4: Latency of the global reset mechanism

Due to the fact that if the global reset mechanism is invoked and activated by a participating node, the other nodes cannot progress in any other part of the system because of overflow risks. Hence, the latency of the global reset mechanism is important to measure in order to find out how long the service of the system will be down in case the mechanism needs to be activated.

The global reset mechanism is expected to get remarkably slower with an increase of participating nodes in the system. This is mainly due to the fact that the mechanism requires all working nodes to state-transition in unison through the mechanism's state-machine in order to reset the corrupted variables. Hence, the slowest node will be the deciding factor of the latency for the global reset mechanism.

The results of experiment 4 is shown in figure 5.5 and indicates a non-linear increase of latency as the number of nodes in the distributed system increases. Due to the fact that the mechanism requires messages to be sent to and received by every non-faulty node in order to execute the reset state-transition in unison, the increase in latency can be explained by the increase of number of messages needed to be sent in order facilitate a transition with more participating nodes.

The experiment results correlate heavily with the anticipated results, indicating that it is most likely the slowest working node which determines the latency of the entire global reset mechanism.

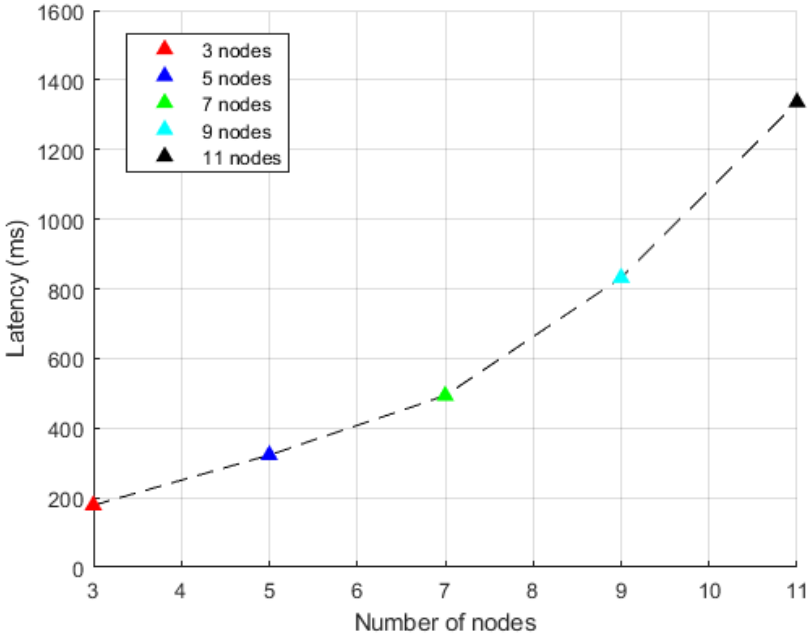

Figure 5.5: The result of experiment 4, showing the latency of the global reset mechanism.

5.1.5 Experiment 5: Throughput of the self-stabilizing state-machine replication

We measure the throughput of the self-stabilizing emulation of state-machine replication in order to get an understanding of the entire system's performance. This is due to the fact that the state-machine replication sits at the top of the stack and utilizes every underlying algorithm to provide its service. The throughput will give us an indication on how often the implemented state-machine will be able to transition.

It is hard to anticipate the performance of the state-machine replication, mainly because there are several underlying algorithms whose performance affects the throughput experiment. However, since the underlying stack seems to drop in performance with the number of participating nodes, we expect similar results from the self-stabilizing emulation of state-machine replication.

The results for the throughput of the self-stabilizing state-machine replication is shown in figure 5.6, there are multiple observations which can be made. Firstly, the performance is degrading when more nodes are introduced into the system. This can be seen in the graph by observing that more deliveries are made during the same time frame when there are fewer nodes in the system. Secondly, the performance gets remarkably lower over time, which can be seen as the lines in figure 5.6 flatten out when running time increase.

The results from the experiment conform to the expectation in general. The throughput get lower with more participating nodes. However, we did not expect the

throughput to be as low for the configurations with 9 and 11 nodes as is shown in figure 5.6. It should be noted that the hardware limitations of the environment can have some effects on the result from the larger configurations, resulting in more extreme results. The main reasons as to why the throughput gets lower over time is because there is a need for more rebroadcasts due to the nature of self-stabilization and due to the fact that the buffers get filled up.

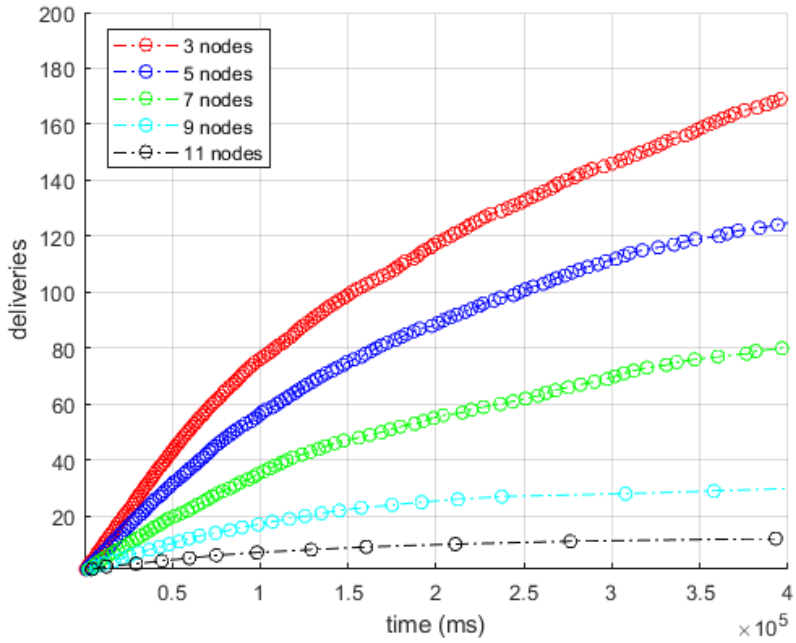

Figure 5.6: The result of experiment 5, showing the throughput of the self-stabilizing emulated state-machine replication. Each circle represents a delivery.

5.1.6 Experiment 6: Crashing of participating nodes

An important aspect of the implementation is the system's ability to handle the crashing of participating nodes. A system which suspends its operation once a single node has crashed cannot be considered fault-tolerant.

The anticipated results from the experiment is for the implementation to continue operating as normal while the number of crashed nodes are less than the smallest required quorum in the system for progress. Once the number of crashed nodes becomes larger than the smallest quorum, the system should come to a halt.

The results from experiment 6 showed that the system was able to operate according to the specifications while the number of crashed nodes remained less than the smallest required quorum in the system for progress. When the threshold was exceeded we observed that the state-machine was not able to come to any agreements on the next state transition.

These results are according to specification of the algorithms in the stack. The

self-stabilizing indulgent zero-degrading binary consensus [38] requires a quorum of functioning nodes in the system in order to complete the two phases which are a part of its operation. Once the number of crashed nodes exceeded the required quorum, the distributed system came to a halt in the phases of the binary consensus and thus the system came to a halt. This is expected since the binary consensus algorithm is at the bottom of the stack meaning that the multivalued consensus and the state-machine replication itself are dependent on it.

5.1.7 Experiment 7: Leader failure

A leader is required in order for the consensus algorithms to function. Hence, if the current leader crashed due to a failure, the system should be able to elect a new leader to continue operating.

We anticipate that once the current leader has crashed, the working nodes should detect that the current leader is unable to respond to the failure detectors messages. This information should also be propagated amongst the working nodes *count*[] array through their messages. Once the number of suspicions on the current leader exceeds the least suspicious node, this node will become the new elected leader of the system.

By the process of suspending the current leader in the system as part of experiment 7, it was possible to validate the system's ability to elect a new leader. Once a new leader had been elected, it was shared amongst all non-faulty participating nodes in the system. This is achieved through the propagation of the *count*[] array which ensures that information about faulty nodes is shared with everyone in the system.

This result is correct in accordance with the description and specification of the self-stabilizing failure detector [38]. Experiment 1 shows the latency of electing a new leader with different configuration of nodes.

5.1.8 Experiment 8: Demonstrating recovery from synthetic memory corruption

We evaluate the system's ability to handle and recover from synthetic memory corruption injections. This evaluation allows us to evaluate the self-stabilizing aspects of the implementation. The algorithms in the stack should all be able to return the system to a legal state of execution in the case of an arbitrary transient fault, which in this experiment is demonstrated through the injection of synthetic memory corruption.

The system should be able to return to a legal state of execution within a finite time window in the case of a synthetic memory corruption which causes the system to illegitimately leave the legal state of execution.

Experiment 8 involved executing a piece of code which injected a synthetic memory corruption into the operation of a non-faulty participating node in the system. More specifically, into the object/instances of the binary and multivalued consensuses and

the failure detector.

When the round number was modified in the active consensus instances, the progress of the consensus agreement was undisturbed. The instance was still able to reach an agreement which was shared amongst all working nodes.

When the *count*[] array was modified in the participating node's failure detector it was handled by the *check*() function. Once the corruption was injected into the operation of the node, the self-stabilizing elements of the algorithm handled and restored the system to a legal state of execution from the previous illegal state which included the synthetic memory corruption.

The response from the system was correct in accordance with the described specifications of the algorithm [38, 39]. Within a finite number of execution rounds, the algorithm was returned to a legal state and able to operate as normal without any further faults.

5.2 Discussion

The results demonstrate that the implementation of our distributed system for self-stabilizing emulation of state-machine replication is able to handle the faults previewed in the fault model (see *section 1.2*). This includes common problems found in distributed systems operating over asynchronous networks, such as packet loss and crashing of nodes. In accordance with those problems, we also demonstrated that our implementation is able to handle arbitrary transient faults, and return the system to a legal state of execution if a transient fault occurs within a finite time window.

With the results from the performance related experiments, we are able to demonstrate how our implementation's performance scales decreasingly with the number of participating nodes in the system. Most of the experiments and their included figures show a trending non-linear or a worst case quadratic increase of latency of the implemented system when the number of participating nodes are increased. This is to be expected since the message-complexity of the algorithms in the stack increases with the number of nodes. An increase of participating nodes correlates to a remarkable increase of required received and sent messages in order to make progress in reaching an agreement on the next state of the self-stabilizing state-machine replication.

Our first research question asked whether the recent research in the area of self-stabilization and fault-tolerance was valid and correct. This question was answered through our implementation and evaluation of the research. The performed experiments and evaluation of the fault-tolerance and self-stabilization of the algorithms in the stack [9, 38, 39] showed that the distributed system which the stack builds is capable of handling the faults in the fault model. In the presence of common distributed system faults such as crashing of nodes, our implementation was able to continue working as long as the quorum requirement was met. Our evaluation also demonstrated that our implementation was able to cope with arbitrary transient

faults and return the system to a legal state of execution once such a fault occurred. With this result, we have provided a demonstration that the recent research in the area is valid and correct and that it in fact can be implemented in practice. Hence, the research could be used in real-world systems in order to reach higher levels of fault-tolerance.

Our second research question asked what the performance of our implementation is, if the first research question turned out to be true. From our evaluation, we are able to observe that the latency of our implemented distributed system grows at roughly a quadratic rate in relation to the number of participating nodes. This can be most distinctively noticed in experiment 2 and experiment 5 which evaluate two key components of the algorithmic stack. A similar growth in latency can be observed in the global reset mechanism which operates outside of the stack which provides the state-machine service. However, for the self-stabilizing failure detector we do not observe the same half-exponential growth in latency, this could be explained by the fact that the algorithm only needs to wait for the fastest required quorum. It should be noted that neither the recent researched algorithms nor our implementation of said algorithms has placed a high focus on performance development. The main focus has been to implement and validate the research in the area of self-stabilization and fault-tolerance through the explained evaluation and experiments.

6

Conclusion

With this thesis, we have provided an implementation of a stack containing recently researched distributed algorithms from the area of self-stabilization and fault-tolerance. Our implementation utilize *self-stabilizing uniform reliable broadcast*, *self-stabilizing failure detector*, *self-stabilizing indulgent zero-degrading binary consensus*, *self-stabilizing multivalued consensus* and a *global reset mechanism* in order to provide a fully self-stabilizing and fault-tolerant emulation of state-machine replication service [9, 38, 39, 37]. The thesis has primarily focused on validating and evaluating the provided research through this implementation.

The provided results from this thesis project can be used to demonstrate the validity and correctness of the research. The experiments conducted within this thesis shows that the independent algorithms as well as the system which the implemented stack provides is able to handle common faults in distributed systems as well as arbitrary transient faults. Our evaluation also shows that the latency of the implementation grows at a half-exponential rate with the number of participating nodes in the system. These results demonstrate that the recent advancements in the area of self-stabilization and fault-tolerance are valid and can be used to increase the fault-tolerance of future distributed systems.

Extensions

As an extension to the work presented in this thesis there is a possibility to improve the quorum system. The extension reduces the probability of the communication links being flooded with the different types of messages (*alive, response, sync...*) sent by the system's algorithms. The improvement can be done by measuring the round trip time for each link-latency between every node present in the system and only re-transmitting messages on the links once every such time frame. This would allow the broadcasted messages to have a chance of reaching the destination node and responding before a new broadcast of the same identical message is made to the link again.

The improvement can be implemented in two ways. (1) It can be improved by using a measured constant of the estimated round trip time of the link-latencies calculated before running the algorithms presented in this thesis. (2) The second approach is to dynamically calculate the estimated round trip time while the system is running. The delay between each broadcast will then be set based on the upper-bound of the

previous time it took to get the responses for the specific messages. Both approaches have pros and cons, the first approach completes the calculations before starting the system's algorithms, hence does not need to spend resources measuring and calculating during the system's operation. However, the second approach which uses more resources during the operation of the algorithm is capable of dynamically changing the rate of which messages are re-transmitted as the status of the links changes over time.

In the current implementation of the system, only the total-order broadcast messages part of the self-stabilizing emulation of state-machine replication is utilizing this improvement. Furthermore, the implementation utilizes the first approach of measuring the upper-bound of the round trip times of the link-latencies before starting the operation of the algorithms. It was purposely decided using this approach during the project because the link-latencies do not change when running evaluation locally.

Bibliography

[1] J. Beauquier, T. Hérault, and E. Schiller. Easy stabilization with an agent. In *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2001.

[2] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[3] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid. A self-organizing distributed and in-band SDN control plane. In *ICDCS*, pages 2656–2657. IEEE Computer Society, 2017.

[4] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid. Renaissance: A self-stabilizing distributed SDN control plane. In *ICDCS*, pages 233–243. IEEE Computer Society, 2018.

[5] A. Casimiro, E. Ekenstedt, and E. M. Schiller. Self-stabilizing manoeuvre negotiation: The case of virtual traffic lights. In *SRDS*, pages 354–356. IEEE, 2019.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

[7] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

[8] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing reconfiguration. In *Middleware Posters and Demos*, pages 13–14. ACM, 2016.

[9] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing reconfiguration. In *NETYS*, volume 10299 of *Lecture Notes in Computer Science*, pages 51–68, 2017.

[10] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing reconfiguration. In *NETYS*, volume 10299 of *Lecture Notes in Computer Science*, pages 51–68, 2017.

[11] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Practically-self-stabilizing virtual synchrony. *J. Comput. Syst. Sci.*, 96:50–73, 2018.

[12] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing byzan-

tine tolerant replicated state machine based on failure detectors. In *CSCML*, volume 10879 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2018.

[13] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Brief announcement: virtual mobile nodes for mobile ad hoc networks. In *PODC*, page 385. ACM, 2004.

[14] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. L. Welch. Autonomous virtual mobile nodes. In *SPAA*, page 215. ACM, 2005.

[15] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. L. Welch. Autonomous virtual mobile nodes. In *DIALM-POMC*, pages 62–69. ACM, 2005.

[16] S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing automatic repeat request algorithms for (bounded capacity, omitting, duplicating and non-fifo) computer networks. *CoRR*, abs/2006.05901, 2020.

[17] S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8):884–900, 2010.

[18] S. Dolev, O. Liba, and E. M. 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.

[19] S. Dolev, T. Petig, and E. M. Schiller. Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. *CoRR*, abs/1806.03498, 2018.

[20] S. Dolev and E. Schiller. Communication adaptive self-stabilizing group membership service. In *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2001.

[21] S. Dolev and E. Schiller. Communication adaptive self-stabilizing group membership service. *IEEE Trans. Parallel Distributed Syst.*, 14(7):709–720, 2003.

[22] S. Dolev and E. Schiller. Self-stabilizing group communication in directed networks. *Acta Informatica*, 40(9):609–636, 2004.

[23] S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mob. Comput.*, 5(7):893–905, 2006.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[25] C. Georgiou, R. Gustafsson, A. Lindhé, and E. M. Schiller. Self-stabilization overhead: A case study on coded atomic storage. In *NETYS*, volume 11704 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2019.

[26] C. Georgiou, O. Lundström, and E. M. Schiller. Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. In *NETYS*, volume 11704

of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2019.

[27] C. Georgiou, I. Marcoullis, M. Raynal, and E. M. Schiller. Loosely-self-stabilizing byzantine-tolerant binary consensus for signature-free message-passing systems. *CoRR*, abs/2103.14649, 2021.

[28] Z. Georgiou, C. Georgiou, G. Pallis, E. M. Schiller, and D. Trihinas. A self-stabilizing control plane for fog ecosystems. In *UCC*, pages 13–22. IEEE, 2020.

[29] R. Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 289–297, New York, NY, USA, 2000. Association for Computing Machinery.

[30] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Comput.*, 53(4):453–466, Apr. 2004.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[32] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[34] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[35] O. Lundström and C. Kou. Self-stabilizing byzantine fault-tolerant state machine replication - rust implementation, experimental evaluation and applications in trucks, 2020.

[36] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing set-constrained delivery broadcast (extended abstract). In *ICDCS*, pages 617–627. IEEE, 2020.

[37] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing uniform reliable broadcast, 2020.

[38] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing indulgent zero-degrading binary consensus. In *International Conference on Distributed Computing and Networking 2021*, ICDCN '21, page 106–115, New York, NY, USA, 2021. Association for Computing Machinery.

[39] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing multivalued consensus in asynchronous crash-prone systems. *European Dependable Computing Conference (EDCC)*, 2021.

[40] O. Morales-Ponce, E. M. Schiller, and P. Falcone. How to stop disagreeing and

start cooperatingin the presence of asymmetric packet loss. *Sensors*, 18(4):1287, 2018.

[41] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 351–360, 2003.

[42] A. Mostéfaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73:207–212, 03 2000.

[43] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[44] T. Petig, E. M. Schiller, and J. Suomela. Changing lanes on a highway. In *ATMOS*, volume 65 of *OASICS*, pages 9:1–9:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[45] J. Postel. User datagram protocol. *RFC*, 768:1–3, August 1980.

[46] I. Salem and E. M. Schiller. Practically-self-stabilizing vector clocks in the absence of execution fairness. In *NETYS*, volume 11028 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2018.

[47] V. Savic, E. M. Schiller, and M. Papatriantafilou. Distributed algorithm for collision avoidance at road intersections in the presence of communication failures. In *Intelligent Vehicles Symposium*, pages 1005–1012. IEEE, 2017.

[48] M. A. Skoglund, T. Petig, B. Vedder, H. Eriksson, and E. M. Schiller. Static and dynamic performance evaluation of low-cost RTK GPS receivers. In *Intelligent Vehicles Symposium*, pages 16–19. IEEE, 2016.

[49] A. Wegener, E. M. Schiller, H. Hellbrück, S. P. Fekete, and S. Fischer. Hovering data clouds: A decentralized and self-organizing information system. In *IWSOS/EuroNGI*, volume 4124 of *Lecture Notes in Computer Science*, pages 243–247. Springer, 2006.

[50] J. Zhang and W. Chen. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Inf. Process. Lett.*, 109(17):1005–1009, Aug. 2009.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY