CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Implementation and Evaluation of Shared Memory Emulation on top of Quorum Reconfiguration

Master's thesis in Computer Science and Engineering

David Brandberg & Henrik Hellström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Implementation and Evaluation of Shared Memory Emulation on top of Quorum Reconfiguration

David Brandberg & Henrik Hellström

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Implementation and Evaluation of Shared Memory Emulation on top of Quorum Reconfiguration

David Brandberg & Henrik Hellström

© David Brandberg & Henrik Hellström, 2022.

Supervisor: Elad Michael Schiller, Department of Computer Science and Engineering
Advisor: Ingvar Andersson, Combitech AB
Examiner: Ahmed Ali-Eldin Hassan, Department

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Implementation and Evaluation of Shared Memory Emulation on top of Quorum Reconfiguration

David Brandberg & Henrik Hellström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The purpose of this thesis is to evaluate the performance of an implementation of a self-stabilizing quorum reconfiguration algorithm by Dolev, Georgiou, Marcoullis, and Schiller[1] both in isolation and when paired with the atomic read/write shared memory emulation algorithm by Lynch and Schvartsman [2]. Furthermore, the correctness of the implementation of the reconfiguration algorithm is validated through the use of invariants and lemmas. The performance of the implemented reconfiguration algorithm and the shared memory emulation algorithm are initially evaluated in isolation in order to get the baseline performance. They are then evaluated when paired together in order to evaluate how the pairing affects the performance compared to the performance of the individual algorithms. The evaluation is done in a distributed setting and evaluated for systems consisting of between three and nine processors. The performance metrics of the evaluation are latency, number of messages sent and delivered as well as the number of full-loop iterations necessary to complete an operation. The results indicate that for this implementation, there is no significant loss in performance when pairing the reconfiguration algorithm and the shared memory emulation compared to having them executed in isolation. These results can therefore be seen as an indication that when implemented in a similar manner, having this reconfiguration algorithm is a viable option for providing quorums to another algorithm in a distributed system. In addition, our results show that the main increase in latency when scaling the number of processors is due to the increase in the data being processed, since the number of full-loop iterations necessary for an operation stays constant, even for larger processor configurations.

Keywords: Computer science, engineering, distributed system, self-stabilization quorum, reconfiguration, shared memory emulation.

Acknowledgements

We would like to give a special thank you to our supervisor Elad Schiller and to Ingvar Andersson at Combitech AB. Without your support this project would not have been possible.

David Brandberg & Henrik Hellström, Gothenburg, October 2022

Contents

1 Introduction **1**
1.1 Fault Model . . . 1
1.1.1 Communication failures and processor failures . . . 1
1.1.2 Self-stabilization . . . 2
1.2 Related Work . . . 2
1.2.1 Reconfiguration Algorithm . . . 2
1.2.2 Shared Memory Emulation . . . 2
1.2.3 Snapshot . . . 2
1.3 Our Contribution . . . 3
1.4 Document Structure . . . 3

2 System Settings **5**
2.1 System Assumptions . . . 5
2.2 System Construction . . . 5
2.2.1 User Data Protocol (UDP) . . . 5
2.2.2 Failure Detector . . . 6
2.2.3 Reconfiguration . . . 6
2.2.4 Shared Memory Emulation . . . 6

3 The Studied Task **7**
3.1 Failure Detectors . . . 7
3.2 Shared Memory Emulation . . . 7
3.3 Reconfiguration . . . 7

4 Description of the Implemented Algorithms **9**
4.1 Shared Memory Emulation . . . 9
4.1.1 Primitive Layer . . . 9
4.1.2 Advanced Layer . . . 10
4.2 Snapshot . . . 12
4.3 Self-stabilizing Quorum Reconfiguration . . . 13
4.3.1 Joining . . . 13
4.4 Reconfiguration Management . . . 14
4.5 Reconfiguration Stability Assurance . . . 16
4.5.1 A Detailed Description of recSA . . . 16
4.5.2 Interaction Between Brute-force and Delicate Reconfiguration 20
4.6 Cooperation Between Reconfiguration Layers . . . 21

5 Implementation **23**
5.1 Development . . . 23
5.2 Self-stabilizing Reconfiguration . . . 23
5.3 Shared Memory Emulation . . . 24
5.4 Snapshot Algorithm . . . 24
5.5 Possible Improvements . . . 25

6 Evaluation **27**
6.1 Research Question . . . 27
6.2 Evaluation Criteria . . . 28
6.3 Experiment Description . . . 28
6.3.1 Validate the Correctness of the Reconfiguration Algorithm . . 28
6.3.2 Establish the Baseline Performance of the Shared Memory Emulation . . . 28
6.3.3 Establish the Baseline Performance of the Reconfiguration Algorithm . . . 29
6.3.4 Establish the Performance of the Shared Memory Emulation and Reconfiguration when Paired . . . 29
6.3.5 Evaluating Correctness in a Failure Prone Network . . . 30
6.3.6 Number of Conducted Experiments . . . 30
6.4 Evaluation Environment . . . 30

7 Evaluation Results **33**
7.1 Evaluating the Self-stabilizing Reconfiguration in Isolation from the Application . . . 34
7.1.1 Number of Iterations . . . 34
7.1.2 Number of Messages Sent and Delivered . . . 35
7.1.3 Operation Latency . . . 36
7.2 Evaluating the Shared Memory Emulation in Isolation from the Self-stabilizing reconfiguration . . . 37
7.2.1 Number of Iterations . . . 37
7.2.2 Number of messages . . . 38
7.2.3 Operation Latency . . . 39
7.3 Evaluating the shared memory emulation paired with the self-stabilizing reconfiguration . . . 41
7.3.1 Number of Iterations . . . 41
7.3.2 Number of Messages . . . 42
7.3.3 Operation Latency . . . 44
7.4 Validation . . . 45

8 Discussion & Conclusion **47**
8.1 Snapshot Algorithm and Global Validation . . . 47
8.2 Validation Limitations . . . 48
8.3 Implementation and Evaluation Limitations . . . 48
8.4 Takeaways and Future Work . . . 49
8.5 Conclusion . . . 50

Bibliography **51**

A Appendix 1 **I**

A.1 Lemmas and Invariants I

1
Introduction

Distributed systems is a field of great importance today due to its benefits compared to centralized solutions, for example, the way resources can be shared and accessed [3]. However, going from a centralized to a distributed systems does not come without challenges. For instance, a quorum of active processors can be necessary to solve problems efficiently in a distributed system. Furthermore, no guarantees on which processors are part of the system in the present, nor will be part in the future, can be made. Therefore, even though the system configuration is changing, quorums that are correct need to be available in the system at all times, something which is made possible through a quorum reconfiguration.

This project studies our implementation of an advanced algorithm for reconfiguration of the system quorum by Dolev, Georgiou, Marcoullis, and Schiller [1], which we refer to as DGMS from now on. DGMS's implementation offers a high degree of robustness against faults, leading to better fault tolerance. Our study focuses on the validation of the correctness proof and the provision of a preliminary performance evaluation of the algorithm. In addition to evaluating the isolated performance of the reconfiguration algorithm, the performance is evaluated when paired with a multiple-writer multiple-reader (MWMR) atomic shared memory emulation by Lynch and Schvartsman [2].

1.1 Fault Model

In this section, the fault model from which the system is designed is presented. This includes different types of failures as well as the concept of self-stabilization.

1.1.1 Communication failures and processor failures

The system on which the algorithms are evaluated is assumed to consist of multiple processors communicating over an unreliable network. This means that messages cannot be assumed to always arrive at all or within a finite period of time. Furthermore, processor failures in which a processor within the system crashes or in another way becomes unresponsive, are also possible and thus covered within the fault model. Even though no guarantee on message arrival can be made, are model does not consider the case when messages arrive in a corrupt state i.e. message corruption is not handled and are not be part of failures from which the system can recover.

1.1.2 Self-stabilization

A self-stabilizing system was described by Edsgar Dijkstra in his 1974 paper, *Self-stabilizing systems in spite of distributed control* [4]. In the paper Dijkstra writes, *"We call a system "self-stabilizing" if and only if regardless of the initial state and regardless of the privilege selected each time for the next move, always at least one privilege will be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves."* In other words, this means that from any arbitrary state the system will converge toward the desired behavior automatically. Furthermore, the converging from an arbitrary state to the desired behavior means that a self-stabilizing system will have the ability to recover from arbitrary faults, which can occur during an execution [5].

1.2 Related Work

To the best of our knowledge, there has been no previous project that has evaluated a system consisting of a similar implementation of the same algorithms as in this project. However, there exists much related work which has built the foundation of this project.

1.2.1 Reconfiguration Algorithm

The paper *RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks* by Lynch and Schvartsman [6] provides an early, less robust version of quorum reconfiguration compared to the one presented by DGMS. In the RAMBO paper, it is used as a way of emulating atomic read/write shared memory objects in a dynamic network setting and thus shares similarities with this project.

1.2.2 Shared Memory Emulation

The paper *Sharing memory robustly in message-passing systems* by Attiya, Bar-Noy, and Dolev [7] is an important contribution since they provided two of the first algorithms which emulated a shared memory in a message passing setting. This meant that from then on it was possible to, through abstraction, use solutions that were previously only applicable on true shared memory systems in message passing ones as well.

1.2.3 Snapshot

The snapshot algorithm used in our project (*Implementing Snapshot Object on Top of Crash-Prone Asynchronous Message-Passing Systems* [8] by Delporte-Gallet, Fauconnier, Raisbaum, and Raynal) can be improved further by introducing a self-stabilizing property to a snapshot algorithm. An alternative to the snapshot algorithm used in this project was presented by Georgiou, Lundström and Schiller [9]. Recent advances in the area of self-stabilizing includes a number of communication abstractions, such as for self-stabilazing crash tolerant systems [10, 11, 12, 13, 14]

and self-stabilizing Byzantine-tolerant systems [15, 16, 17, 18, 19]. Earlier work in the area of self-stabilizing communication abstractions included group communications [**DBLP:journals/tpds/DolevS03**, 20, 21, 22, 23], and control planes [24, 10, 25, 26], to name a few.

1.3 Our Contribution

We provide a preliminary implementation of an important component for fault tolerant distributed systems—a RUST implementation of a self-stabilizing quorum reconfiguration using the DGMS algorithm. In addition to the safety benefits provided by RUST, our implementation focuses on the validation of the DGMS algorithm via the invariants derived from the correctness proof available at [1]. Furthermore, our contribution also includes a preliminary performance evaluation through the implementation of a MWMR atomic shared memory emulation using the Lynch Schvartsman algorithm [2] and a network using up to nine system processors. The result of this evaluation shows that there is little to no loss in performance when pairing reconfiguration and shared memory emulation, compared to running in isolation, with the largest penalty in latency being 4.1%. Moreover, the conducted validation showed no sign of faulty operation or results, which strengthens the evidence for the reconfiguration algorithm being designed and implemented in a correct manner.

1.4 Document Structure

The document is structured with an overview of the system settings after the introduction. chapter 2 contains an overview and system assumptions. This is followed by chapter 3, which explains the different tasks which need to be completed for the system to function correctly. In chapter 4, an explanation of how the different algorithms work, and in chapter 5, a more detailed explanation of how the algorithms were implemented is covered. In chapter 6, the reason why experiments were conducted as well as how the algorithms were evaluated is covered. In chapter 7, the results from all experiments are displayed and lastly, in chapter 8, a discussion about the project is available as well as a conclusion.

2

System Settings

The algorithms implemented in this project have to be implemented upon an already existing system with a certain specification. Thus, for the algorithms to function correctly, a number of assumptions about the system on which it is built have to be made.

2.1 System Assumptions

The underlying system consists of a bounded number of processors with unique identifiers. These processors communicate via a fully connected network, meaning that all processors have the ability to send a message to another processor directly, without interference from another member of the network. The network has no bound or guarantees on potential communication delays. The system is also assumed to be asynchronous and thereby no global synchronized clock is available for the processors to use. Both processors and links can suffer from arbitrary failures, thus algorithms implemented should be able to handle and work even in the presence of such failures. In order to detect processor failures, a temporally reliable failure detector is available within the system and can thus be used by other algorithms. Packet losses are considered to be a potential fault, however, packet corruption will not be covered or considered in the scope of this project.

2.2 System Construction

The system can be described as in Figure 2.1, consisting of a number of layers working together to form the complete system. The higher layers depend on the services provided by the layers below, to ensure correct functionality.

2.2.1 User Data Protocol (UDP)

UDP is used to send messages between two processors with Internet Protocol (IP) as its underlying protocol. UDP has the benefit of sending messages with little overhead. However, in order to be able to provide message transmissions with minimum overhead, it does not provide any guarantees of delivery or protection against duplicates [27].

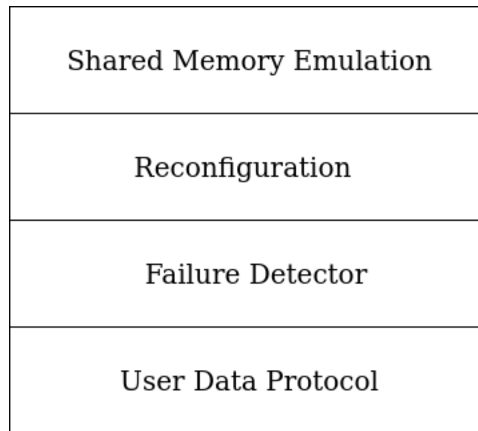

Figure 2.1: Figure depicting the system architecture

2.2.2 Failure Detector

The failure detector presented in the paper [1] provides the reconfiguration algorithm with a view of which processors are not responsive and thus need to be excluded. For the failure detector to be able to get this view, it monitors the rate of other processors' messages and based on this creates a view of which processors are still active.

2.2.3 Reconfiguration

The reconfiguration algorithm is used to provide quorums to the shared memory emulation. It uses the lower layers (see Fig 2.1) to provide this service [1]. For a more detailed description see chapter 4.

2.2.4 Shared Memory Emulation

The shared memory emulation is the top application and makes use of all the services provided by the algorithms below it. The shared memory emulation can be used by other applications which need to work in a shared memory setting, even though the system is a message passing one [2]. For a more detailed description of the shared memory emulation see 4.

3
The Studied Task

In this chapter, three tasks that were studied and form the basis of this project are explained in an overview.

3.1 Failure Detectors

The idea behind a failure detector is to give non-faulty processors the ability to detect if another processor has become unresponsive or has crashed. There are two common ways of building a failure detector. The first being active monitoring, where messages are actively sent to other processors and waiting for a response. The other way is passive monitoring, when the failure detector make use of messages being sent and received by other applications. Relying on other applications can however lead to problems if the message frequency can not be guaranteed. What both of these techniques have in common is the need for a timer and a set timeout limit to detect whether a processor has crashed or not. The use of a timer has its limitations when used in an asynchronous setting since a processor can only suspect that another processor has become unresponsive or crashed. This is due to the lack of a guarantee on when a message will arrive from another processor. Therefore, it could be the case that a processor is still alive even though it has been deemed crashed and excluded by another processor [3].

3.2 Shared Memory Emulation

Through observations done during research related to distributed systems, it has been concluded that in several cases, it is easier to construct algorithms that are able to tolerate timing delays as well as component failures when making use of a shared-memory model as opposed to message-passing. Therefore, it can be of great importance to be able to use an algorithm constructed for a shared-memory model in a message-passing paradigm. Hence, an emulator was implemented based on an algorithm presented by Lynch and Schvartsman that emulates shared memory in a message-passing system [2].

3.3 Reconfiguration

One way of increasing the level of fault-tolerance and availability in a system that provides services to clients is to make use of quorum configurations. Quorum config-

urations are based on accessing intersecting sets of processors when serving requests from clients. The intention of this is to make sure that every action taken is visible to all other clients, even in the presence of processors exhibiting faulty behavior. For example, by ensuring that every pair of write- and read-sets have one processor that is part of both of the sets, it is possible to make sure that after a write-action has been performed, any subsequent read-action requested by a client will access the latest value in the system. However, it is not uncommon that the processors that are part of the quorum alternate over time, as processors crash or become unable to communicate due to changing network topologies. This can generally be solved by having a new, active processor join the quorum through the initiation of a reconfiguration process, that creates a new quorum based on a defined set of participants [1][2].

4

Description of the Implemented Algorithms

4.1 Shared Memory Emulation

When implementing the shared memory emulation algorithm that is presented in the paper by Lynch and Schvartsman, a decision was made to make use of the same communication abstractions as for the other implemented algorithms, i.e. shared memory and quorum reconfiguration, rather than developing new ones. Hence, the implementation differs in some specific aspects from what is presented in Appendix A of the paper.

The emulation algorithm consists of two different layers. The first layer is referred to as the primitive, which handles low-level communication by propagating basic messages as well as acknowledgments. The advanced layer takes care of the actual emulation [2][28].

4.1.1 Primitive Layer

Any action taken by the primitive layer is initiated by a call from the advanced layer to the *submit*-function. In this call, the advanced layer specifies the following values:

- m, which is the messages that are to be communicated by the primitive layer
- c, which is the condenser function that is to be used. The condenser function takes a vector of responses as input, which is received from its own advanced layer as well as the advanced layer of other processors. Based on this vector, the condenser function returns a single value. One example of a condenser function is to get the register-value with the highest lexicographical value from a number different processors.
- s, which indicates whether the read or the write quorum is to be used
- id, which is a unique operation-id that is used to identify the operation during the processing of the request

Upon receiving a call to the *submit*-function, the values specified are inserted into a list structure, where they can be accessed using the operation-id. Furthermore, the

specified message together with the id is communicated using broadcast to every processor in the quorum, specified in the *submit*-call. When a recipient receives this tuple, it proceeds to make a call to the *deliver*-function belonging to the advanced layer, specifying the message, the operation-id as well as the sender of the tuple [2].

After some time, the primitive layer of the receiver will receive a call to the *ack*-function from the advanced layer. The function call will have a tuple as an input, consisting of a response value and the operation-id of the original *submit*-call. Subsequently, the primitive layer will send the received tuple to the processor associated with the operation-id. When the original sender receives this message, it will save the response value in a vector, where it will be the entry at the index corresponding to the processor-id of the sender [2].

When the processor which handled the original *submit*-call has received an acknowledgment from the primitive layer of every processor that is in the quorum, it makes use of the specified condenser function to get a single response that is based on all the values attached to the acknowledgments. A tuple, containing the response and the operation-id, is then sent to the advanced layer. All the information associated with the operation-id in the list-structure is then removed [2].

4.1.2 Advanced Layer

The advanced layer can be described as a state-machine, where the actions that can be taken are strictly based on the current state of the processor. The possible states are query-ready, query-active, prop-ready, prop-active, and prop-done. The initial state of every processor is set to query-ready. The algorithm can furthermore be divided into two different stages, each one consisting of a broadcast together with the reception of an acknowledgment from every receiver. The first stage is *query*, where the processor initiating the operation queries the quorum for the values and the tags of every register. The second stage is *propagate*, which updates the registers in the quorum [2].

The advanced layer makes use of two additional variables for the read- and write operations:

- $prop-val$, which is either the value to be written to the registries or the latest value contained in the registries, depending on the operation
- $prop-tag$, which in the case of a write-operation is the new tag that is to be attached together with the value, or in the case of a read-operation the largest tag read from all the registers in the quorum

In order to initiate a read- or write-action, the advanced layer needs to receive either a *read*-call or a *write*-call from a client. In the case of a *read*-call, a tuple consisting of a value that indicates that the action to be taken is a read, together with the id of the caller is appended to the end of a request-queue. In the case of a *write*-call however, the received input also consists of the value that is supposed to be written,

which is attached to the tuple that is appended to the request-queue [2].

Given that the request-queue is not empty and that the current state of the processor is query-ready, the advanced layer makes a *submit*-call to the primitive layer of the type *query*, indicating that it wants to make a request to all the processors that are in the read-quorum, in order to gain knowledge of the largest tag of all the registers [2].

Upon receiving a call to the *deliver*-function from the primitive layer, the advanced layer proceeds to create a tuple that is appended to the acknowledgment-queue. This tuple consists of a value that indicates that the acknowledgment is in response to a query, the actual value of the register, the tag that indicates the latest write-action taken by the processor with respect to the register as well as the operation-id that is provided as input to the *deliver*-call by the primitive layer. Subsequently, upon noticing that the acknowledgment-queue is not empty, the advanced layer makes a call to the *ack*-function of the primitive layer, by providing the tuple that was previously appended to the queue as input [2].

As a response to receiving the acknowledgments from every processor in the read-quorum, the primitive layer will make a call to the *respond*-function of the advanced layer, providing the output of the condenser-function that was attached in the original call to the *submit*-function that is associated to the operation-id, and the mentioned operation-id, as input. In the case that the current request is a *write*-call, the variable $prop - val$ is assigned the value that is supposed to be written and the variable $prop - tag$ is set to be a tuple that consists of the local sequence value after it has been incremented, and the id of the processor. However, if the current request is a *read*-call, $prop - val$ is set to be the value that is provided in the output, which is the value read from the register that has the largest tag of all the registers. Furthermore, $prop - tag$ is set to be this tag. In both cases, the state of the processor is changed to be prop-ready [2].

Upon noticing that the state is prop-ready, the advanced layer makes a call to the *submit*-function of the primitive layer. To this function call, a tuple is attached as input together with a value that indicates that the *submit*-call is of the type *propagate*. This tuple consists of the value assigned to $prop - val$, the value assigned to $prop - tag$, a value that indicates that the quorum that is to be used is the write-quorum as well as an operation-id. Subsequently, the state of the processor is changed to prop-active [2].

Upon receiving a *deliver*-call from the primitive layer as a response to a *submit*-call of the type *propagate*, the following procedure is taken. Attached to the *deliver*-call is the operation-id, together with the value and the tag that it got from the previous *submit*-call. The advanced layer checks if this attached tag is larger than the local tag, based on lexicographical ordering. If this is the case, it proceeds to change the local value and the local tag to the values that are contained in the *deliver*-call, Whether or not this is the case, it finally adds a tuple to the acknowledgment-queue.

This tuple consists of a value indicating that the *deliver*-call has been handled, together with the operation-id that was provided with the input. As previously described, the tuples appended to the acknowledgment-queue are handled as soon as the advanced layer notices that this queue is not empty [2].

As a result of receiving acknowledgments related to the original *submit*-call of the type *propagate* from all the processors in the quorum, a *respond*-call is sent to the advanced layer. This *respond*-call does not contain anything besides the original operation-id. As a result of receiving this call, the state of the processor is changed to prop-done [2].

In the case that the state of the processor is prop-done and that the current request is of the type *read*, the advanced layer proceeds to remove the current request and change the state of the processor back to query-ready, indicating that a new request can be handled. Furthermore, a call is made to the client of the type $read-confirm$, where the value assigned to the variable $prop - val$ is attached. The same action is also done in the case that the current request is of the type *write*, besides that the call to the client is instead of the type $write - confirm$ and that no value is attached.

4.2 Snapshot

The snapshot algorithm used is one developed by Delporte-Gallet, Fauconnier, Raisbaum, and Raynal and can be found in the paper *Implementing Snapshot Object on Top of Crash-Prone Asynchronous Message-Passing Systems* [8]. A snapshot algorithm like this is used to get a view of the system at a certain point in the execution. For this to be possible it is necessary for the processors to coordinate in order to make sure that no processor changes its state or that any state transitions are recorded by the other processors. The algorithm is also designed to be non-blocking and thus should not interfere with other processes.

The algorithm consists of two types of operations, one known as a write operation and the other known as the snapshot operation. The snapshot operation works as follows:

Algorithm 1 Snapshot Algorithm

1: In an initial step every processor keeps a record of the state of all other processors in the system as well as a sequence number denoted ssn which is used to keep track of which received message is associated with which sent message.
2: Once a processor wishes to take a snapshot, it records its current view of the system. It then increments its ssn sequence number before broadcasting its view as well as its ssn to the other processors.
3: Upon receiving the broadcast, the other processors check if the received view is more recent than the one it has, if so is the case, it will update its view of the system and respond with the updated view along with the ssn sequence number of the broadcast it just received.
4: The processor which initiated the snapshot will then wait until it has received a response from all other processors. Once it has received all responses it checks if all other processors have the same view of the system and if so it knows that at that instant its view of the system is the correct one and a snapshot has been taken.
5: If its own view does not correspond with the views of the others, the broadcast will be re-transmitted with a new updated view of the system and ssn sequence number. This will be repeated until its view matches the view of all others and a snapshot can be taken.

The write-operation works similarly to the snapshot-operation, with the intention of only updating the view of the system. When a processor intends to update its system view and not take a snapshot, it sends a write request together with the view it currently has of the system. The processor will then wait until it receives a response from the other processors in the system. This response contains the view of the respondent and if it had a more updated view than it received, the response will contain that view. In case its view was older than the one received in the write request, it will update to the view it received and respond with that. The processor which initiated the write operation collects the responses from the other processors and based on these updates to the most up-to-date view available.

4.3 Self-stabilizing Quorum Reconfiguration

In this section, the three different parts that make up the self-stabilizing quorum reconfiguration algorithm by DGMS are covered.

4.3.1 Joining

One important feature of the self-stabilizing reconfiguration algorithm is the ability for a new processor to join and become a participant and thus be accepted by the other processors in the configuration. In order to allow this, a joining mechanism is available providing a mechanism for both the members of the configuration to accept new members, as well as for the processor which intends to become a participant to join. A description of the different steps involved when joining can be found in

algorithm 2.

Algorithm 2 Joining Mechanism

1: The joining processor begins by noting which configuration members its failure detector has deemed to be alive. This is done to indicate from which processors the joining processor has to be approved to join.
2: In the next step the joining processor sets its own state into a predefined default state in preparation to be able to join when approved by the other processors that are currently part of the configuration.
3: In order to become a participant, the joining processor repeatedly checks that no reconfiguration is currently taking place as well as if it has received permission to join from more than half of the members of the configuration, which when it first tries to join it naturally has not yet got.
4: If the processor has not yet been accepted by the configuration members, it will send a join-request to all members of the configuration. It is first when the joining processor has been accepted to join that it will stop sending the requests to the configuration members.
5: The joining processor will then continuously check for responses to its join-request while the configuration members will check for any incoming join-requests. If a join-request is received, the responding processor from the configuration sends a response that the joining processor can join if no reconfiguration is currently taking place. The response contains the okay to join as well as the state the joining processor should set its own state to when joining.
6: When the response from one of the configuration members is received by the processor which wants to join, the joining processor will update its view of which processors it has been approved by.
7: Finally, when the joining processor has been approved by more than half of the configuration members, it will be able to join with the state it got as a response to its join-requests from the configuration members.

4.4 Reconfiguration Management

The reconfiguration management layer is tasked with initiating a reconfiguration in two different cases. The first case is when the configuration majority has been lost and the second is when it is predicted from a majority of members that a reconfiguration will be necessary in order to preserve the correct behavior. The reconfiguration management is however not responsible for the reconfiguration itself but instead initiates a reconfiguration by making a call to the reconfiguration stability assurance layer which then completes the reconfiguration. A description of the steps taken to complete this can be found in 3

Algorithm 3 Reconfiguration Management Layer

1: A processors executing the reconfiguration management must be a participant in order to take any of the steps below.
2: The processor will then read the current configuration as well as record that currently it sees that there is an active majority.
3: The processor then looks at the configuration it had recorded from the previous round and compares this to the one it read in the previous step. If it has changed it will reset its local values corresponding to processors active according to its failure detector. After the reset, the processor will assume that non of the processors has reported that there exists no majority.
4: If the reconfiguration stability assurance layer can allow for a new reconfiguration to occur, the processor will check if, from its view, provided by its failure detector, it can see a majority is available.
5: If it fails to see a majority, and a trusted group of processors which is known as the *core* also fails to see a majority, a call to the reconfiguration stability assurance layer for a reconfiguration with a proposed set of processors is made. Furthermore, a reset of variables containing the information whether there exists a majority is done.
6: There is an application-specific function used to predict if a reconfiguration should be made and if there is a majority of members who also predicts that a new reconfiguration is necessary, the reconfiguration stability assurance layer that a reconfiguration is informed that a reconfiguration is necessary and a proposal for a new configuration set.
7: The information regarding the view of whether a reconfiguration should be made is broadcasted between the participants and once a message is received variables can be updated so the view of the other participants' view of the system is up to date.

4.5 Reconfiguration Stability Assurance

The reconfiguration stability assurance layer (recSA) is tasked with making sure that all participants in the quorum always share the same identical view of which processors are currently part of the configuration. The technique used to achieve this can be divided into two different methods, the first is the brute-force stabilization method which is tasked with handling the presence of stale information. Stale information is information that for some reason is invalid or faulty and thus needs to be handled by the system through a reset and a reconfiguration. The second method is a delicate reconfiguration replacement which updates to a new configuration when a processor proposes a new configuration without the presence of stale information. When a new proposal has been received, the processors will through the delicate reconfiguration go through three phases in a coordinated fashion in order to result in the same configuration across all members. A detailed explanation of how the reconfiguration stability assurance layer functions can be found in 4.5.1, together with 4 and 5.

4.5.1 A Detailed Description of recSA

Every processor, p_i, has a number of local variables that are modelled as arrays, where the entry at index i defines p_i's own value and entry j is the value that p_i received most recently from process p_j [1][8].

The first variable is the configuration variable, $config$, which at entry i stores what p_i considers to be the current configuration and at entry j stores the value that p_i received most recently from p_j.

The second variable stores the values received from every processor's failure detector, FD, where entry i is the value received from p_i's failure detector and entry j is the most recently received value from p_j, which in turn received the value from its failure detector. Based on the values contained in $config$ and FD, p_i constructs another variable, called $FD.part$. The entry at index i stores the set of IDs that are contained in the i'th entry of FD, while also being considered a participant according to $config$. The entry at index j however is the latest value received from p_j [1][8].

The third variable, prp, contains proposals for the delicate reconfiguration mechanism. The value at index i stores p_i's progress in the reconfiguration, while index j stores the most recently received value from processor p_j. Every proposal contains two different parts, the first one being the current phase of the reconfiguration and the second one being the proposed new configuration. In the case that no delicate reconfiguration is currently taking place from the view of p_i, the entry at all indices has the phase value set to 0 and the reconfiguration set is empty [1][8].

The fourth variable, all, is used to indicate whether its notifications have been observed by other processors in the system. Every entry in the array contains a boolean. The entry at index i indicates whether all processors p_i considers being

active have observed p_i's current proposals, while also having the same proposals stored. The entry at index j on the other hand stores the last value received from p_j.

The fifth variable, *allSeen*, stores the IDs of processors from which p_i has noticed a completion of the criteria for the current phase. It is used to ensure correct functionality [1][8].

The sixth and final variable is *echo*, which contains some specific values received by other processors. The entry at index i is a tuple, which contains the entry at index i of $FD.part$, *prp* and *all*. The entry at index j however contains the most recently received value from p_j, which in turn is the most recent value that p_j has received from p_i. It can hence be described as a variable containing values that are echoed from other processors in the system [1][8].

A degree is an abstraction that takes both the current phase of the processor, as well as the value of the variables *all* and *allSeen* into consideration. The degree of p_i is equal to the phase of the entry at index i in *prp* times 2, plus 1 if either the value at the entry at index i in *all* is true or if p_i has observed that another processor p_j that is included in *allSeen* and is one phase ahead of p_i [1][8].

Algorithm 4 Detailed Description of the Stability Assurance Layer: Part 1

1: The stability assurance algorithm is encapsulated in a do-forever loop, a very common approach used in self-stabilizing algorithms. In this loop, every processor receives information from other processors, processes this new information, and proceeds to send its local variables to every other processor in the system.

2: In the first line, the algorithm starts off by p_i labeling processors that are considered to be non-active by the failure detector as non-participants, while also setting their configuration proposals to be the default-proposal

3: In the second line, p_i proceeds to check if the processors have a correct degree in relation to all the other processors that are considered to be participants and its current configuration. If that is the case, the set of the proposal at entry i in *prp* is assigned to be the set contained in the proposal with the highest lexicographical value in *prp*. Furthermore, if the phase of the entry at i in *prp* is 0 and there exists a proposal in *prp* that has phase 1, but no proposal with the phase being equal to 2, the variable $allSeen$ is emptied and the phase of the entry at index i in *prp* is set to 1.

4: In the third line, p_i checks if there exists any stale information in its local variables. If that is the case, it informs the other processors in the system by setting the entry at all the indices in $config$ to be empty.

5: In the fourth line, p_i sets the value of the entry at index i in *all* to be equal to true if p_i has a correct degree in relation to all other processors that are in the entry at index i in $FD.part$ and all the other processors contained at index i in $FD.part$ have echoed back the same value as p_i currently holds in the entry at index i of both $FD.part$ and *prp*. This is done using the values at the entry at index i of *echo*.

6: In the fifth line, p_i adds all the IDs of the processors in $FD.part$ for which the entry in *all* is equal to true, into $allSeen$.

Algorithm 5 Detailed Description of the Stability Assurance Layer: Part 2

1: In the sixth, seventh and eighth line, p_i checks if the phase of all processors part of $FD.part$ is equal to 0, indicating that there is no delicate reconfiguration currently taking place. If this is true, it proceeds to check if there are any observable conflicts in the entries in the $config$ variable. If there is any conflict, all entries in $config$ are set to be empty. Finally, if for p_i the entry at index i is empty but all the processors included in the set contained in the entry at index i in FD have the same view of which processors are active, i.e. all of their entries in FD are the same, then the current configuration is replaced by the set at entry i of FD.

2: In the ninth, tenth and eleventh line, p_i checks if there are any delicate reconfiguration currently taking place. In that case p_i proceeds to check if the proposal of the entry at index i of *prp* has a phase that is 2, and the value of *all* at index i is true. If this is also the case, it changes the current configuration by assigning the value at index i of $config$ to be equal to the set in the proposal of the entry at index i of *prp*. Furthermore, it checks if the value of the entry at index i in *all* is equal to true and every processor included in $FD.part$ is also included in $allSeen$. If this is true, and p_i has a correct degree in relation to all the other processors that are in the entry at index i in $FD.part$ and all processors part of the set at index i in $FD.part$ have echoed back the same values as p_i stores at index i in $FD.part$, *prp* and *all*, p_i proceeds to increment the value of the phase of the proposal at index i of *prp*, changing the value from either 1 to 2 or from 2 to 0 (the incrementation from 0 to 1 is handled previously), while also assigning an empty value to the entry at index i of $allSeen$ and the value at index i of *all* to false. Moreover, in the case of changing the phase from 2 to 0, it also empties the set of the proposal at index i in *prp*.

3: At the end of each asynchronous round of the algorithm, if p_i considers itself to be a participant, it broadcasts a number of its own values to a selected group of the processors. These are the processors that are part of the set at index i of FD. In the broadcast message sent to processor p_j, processor p_i includes the values at index i of the following variables: FD, $config$ and *prp*. It also sends a boolean that is true if either the entry at index i of *all* is true or if p_i has received indications of another processor having a proposal with a phase that is ahead of p_i's own proposals phase by one step. Furthermore, the values sent to p_j also includes the entries stored at index j of the following variables: $FD.part$, *prp* and all. These last values are included to ensure an echo-mechanism that is used as a way of acknowledging that a processor has received another processor's current state

4.5.2 Interaction Between Brute-force and Delicate Reconfiguration

The brute force reconfiguration- and the delicate reconfiguration method are both parts of the stability assurance layer and provide the system with a reconfiguration. However, how and when they do a reconfiguration differentiate them from each other but these differences are what make them a powerful method of handling reconfigurations when combined.

The brute-force stabilization functionality is activated in the presence of stale information, which can be classified into four different types. The first type considers the case when there exist notifications of configuration replacements in the system that are considered invalid. The second type encapsulates the case when there exist two active processors that do not share the same view of the current configuration. The third type deals with information regarding a delicate replacement being out-of-sync. Finally, the fourth type checks if there are no participants in the configuration that are active. If stale information of any type is detected, the processor assigns a specific value to its configuration variable in order to signal this occurrence to the rest of the system. Subsequently, by making use of the information provided by the failure detectors, every active processor in the system attempts to agree on a single set of processors, which ends up becoming the new configuration. It is worth noting that this process makes it possible for processors trying to join as well as previous participants, to become participants in the new configuration.

By making sure that if present, stale information will be removed through a reset and reconfiguration by the brute force reconfiguration, the delicate replacement can start without having the risk of encountering these types of faults.

The delicate configuration replacement is used when a processor proposes a new configuration to replace the current one without the presence of stale information. It is initiated through a call from the reconfiguration management process and is divided into three different phases. If no stale information exists in the system, the processors intervene between these phases uniformly, meaning that all processors must reach the same phase before a processor can proceed to the next phase. In the first phase, phase 0, the processors continuously make sure that there is no stale information present. Upon receiving a new configuration proposal, a processor proceeds to enter phase 1. In phase 1, all the participants jointly and in a deterministic manner agree on a single proposal, which is the proposal with the highest lexicographical value. When this is done, and no other proposals are contained within the system, it is possible for a processor to proceed to phase 2, where the current configuration is replaced with the proposed one. Once this is done by all participants, and they thereby agree on the configuration, all participants subsequently return to phase 0 to look for stale information.

Having a robust stability assurance layer which the other layers communicate with in order to establish any changes to the configuration is beneficial for avoiding po-

tential faults arising from the other layers. An example is the joining layer where a call to the *participate*() function is made after it has been checked with the application and members of the configuration. This call does not however mean that the processor becomes part of the configuration immediately, instead, it only makes a proposal of a new configuration with itself as a participant. After the call has been made, a full reconfiguration must be done before the new processor can be considered a member of the configuration by the other processors. This adds the robustness and fault tolerance given by the stability assurance layer to operations stemming from the join mechanism.

4.6 Cooperation Between Reconfiguration Layers

As mentioned previously, the reconfiguration can be divided into three different parts, the stability assurance layer, the reconfiguration management layer, and a joining layer. These three layers are then combined to form a complete reconfiguration algorithm that can work in tandem with an application, in this case, a shared memory emulation application. In this section, the focus will be on how the different parts are communicating and how their different services work together to achieve the task of providing a self-stabilizing reconfiguration. Furthermore, how the brute-force and delicate replacement work together to form a robust stability assurance layer is also covered.

The first combination is between the joining mechanism and the reconfiguration stability assurance layer. Here, the joining mechanism is the one tasked with handling processors who wish to become participants. When the joining mechanism considers members to have permission to join and this has been communicated between more than half of the active members, a call to the function *participate*() is made by the joining processor. This function is called by the joining mechanism but is executed and has its effect in the stability assurance layer and can thus be seen as a communication between the two layers. When the *participate*() call is made, first the stability assurance layer makes sure that no reconfiguration is currently taking place. If that is the case the configuration according to the processor's own view of the system will be updated with it being part of the configuration. However, this does not mean that all other processors will have the joining processor as part of their configuration. For this to happen a reconfiguration must first be made so that all processors can agree upon a single configuration.

The communication between the stability management layer and the stability assurance layer is done through a call on the function *estab*(). The *estab*() function is called upon when the management layer has a new configuration it wants to establish in the system. Once a call has been made, the management layer sends the *estab*() function with its own failure detector's view of who the participants are. When the *estab*() call reaches the stability assurance layer, the stability assurance layer uses the set it was given as a parameter in the *estab*() function and updates its local entries to signal that it has a new configuration which the other processors should adopt via a reconfiguration.

The application, in this project the shared memory emulation algorithm, can have an important role in determining when a reconfiguration is necessary. The application can through the use of the $evalConfig()$ function, communicate to the management layer that it predicts that a reconfiguration is necessary. How the $evalConfig()$ function is implemented is application specific, meaning that different applications have different criteria for when it needs a reconfiguration. In this project, the $evalConfig()$ function has been implemented to ask for a reconfiguration once more than half of the members are not trusted. However, in other applications the $evalConfig()$ can for instance be implemented to trigger a reconfiguration based on certain network criteria.

When a processor wishes to join and become a participant it is not only the joining mechanism and stability assurance layer involved. The application must also approve that a new processor is joining. This is done through the function $passQuery()$, which communicates between the application and the joining layer. If the application can accept a join, the $passQuery()$ function will return $true$, otherwise $false$.

In order for the other layers and applications to know the current status of the stability assurance layer, the two functions $allowReco()$ and $getConfig()$ can be used. The $allowReco()$ function is used by the other layers to check if a reconfiguration is currently taking place and if interactions with the stability assurance layer, therefore, need to be postponed. The $getConfig()$ function can also be used by the other layers and gives the ability to get a view of the current configuration, something that is crucial for the other layers to be able to complete their respective tasks.

5
Implementation

In this section, the development process will be explained as well as how the evaluation was done. The choices made will be discussed and the thought process behind the implementation of the algorithms.

5.1 Development

The entire project has been developed using the programming language Rust. The reasoning behind this is twofold, the first being that the project is a continuation of a previous master's thesis project, and implementation of many core components and a package manager was thus available. The other reason was that Rust is a well-supported programming language designed with performance in mind, which has been beneficial throughout the development process since the evaluation was done on machines with limited computational resources. Using Rust also meant that a broad range of libraries could be used which spared both time and resources. The Rust implementation was tested locally during the development phase with virtual processors running in separate threads. When the implementation was finished and ready to be evaluated on the SNIC SSC network (for information about SNIC SCC see 6.4), executable files were built using a special build tool known as musl. This was done since the virtual machines on the SSC network were not equipped with the latest version of GLIBC and it was possible to create executables that did not depend on one of the later versions GLIBC to run if musl was used. How using musl instead of the standard build tool affected performance is something that was not possible to examine within the scope of this project.

5.2 Self-stabilizing Reconfiguration

The self-stabilizing reconfiguration algorithm can be divided into three different parts, stability assurance, management, and joining. Initially, there existed an implementation of a fundamental part of the stability assurance layer, namely the phase transition, that had been developed by previous students. Although it was not possible to directly extend this work to an implementation of the assurance layer, it provided good insights into how an implementation could be structured. Thus, the stability assurance layer was a reasonable starting point for the development of the system. In parallel with the development of the algorithm, invariants were constructed based on the available pseudo code and proofs from the paper [1]. These invariants served as a way of broadening the understanding of the algorithm

as well as a way of ensuring both correct behavior and proper functionality of the implementation. The way the implementation was validated was through a debugger, that was based on the aforementioned invariants and executed concurrently with the algorithm. Once the implementation had been validated and seemingly provided correct behavior and functionality, the management, as well as the joining mechanism, was developed.

5.3 Shared Memory Emulation

The shared memory emulation by Lynch and Schvartsman [2] was implemented as the application layer of the system. It was developed when the self-stabilizing reconfiguration was able to provide quorums since it relies on the quorums provided by the reconfiguration. The emulation paper was not as strict in its pseudo code compared to the reconfiguration algorithm. Thus, some decisions had to be made regarding the implementation, for instance, the condenser function (see 4.1 for more information). In order to limit the implementation complexity of the emulation application, a decision was made to implement it in the same setting as the reconfiguration algorithm, even though this had a negative effect on the performance. This meant that messages related to the emulation were piggybacked on top of broadcasts made by the reconfiguration algorithm and that no additional threads were used.

The shared memory emulation algorithm is separated into two different layers, both handling different tasks. The implementation made in this project does not make use of explicit layers but instead makes use of message queues to transition between states when handling client requests.

5.4 Snapshot Algorithm

The snapshot algorithm was implemented in order to give a coordinated image of the state of all processors at an instance in time. It was implemented in the same setting as the reconfiguration algorithm since it needs direct access to the reconfiguration state and its messages being piggybacked on top of reconfiguration messages. The reason for having the messages piggybacked is the same as previously mentioned for the emulation algorithm. Furthermore, in order to limit the complexity of the implementation, the write-operation is never used. This is because, in the case of the reconfiguration algorithm, there is no need to update state transitions since they are fast and few. Therefore, when taking a snapshot, a single call to the snapshot-function is made as well as a coordination attempt. Because of the previously mentioned characteristics of the state transitions of the reconfiguration, it is necessary to slow down the system when attempting to take a snapshot. This allows for the snapshot algorithm to coordinate.

5.5 Possible Improvements

During the implementation of all the algorithms, the intention has been to make the code as easy to follow as possible. As a result of this, there are several sections in the implementation where there has been a trade-off between availability and performance. Due to this, possible future work could focus on reducing the complexity of the main loop of the implementation, e.g. decreasing the number of accesses to local variables as well as removing redundant iterations during the execution. Furthermore, the performance of the implementation could also possibly be increased by breaking the large main loop into smaller components, that could be run in different threads. In the current implementation, messages related to the shared memory emulation are piggy-backed on messages related to the reconfiguration algorithm. Another way of implementation would be to make use of direct messaging together with interrupt-calls, which could possibly shorten the latency of read/write operations.

6
Evaluation

Evaluation is a key part of this project and this section contains, a description of what will be evaluated as well as the criteria set for the system. Furthermore, it provides a description of the experiments used to evaluate the system.

6.1 Research Question

This project aims to answer whether the self-stabilizing reconfiguration algorithm presented in [1] can be used as part of a system that provides shared memory emulation based on the quorums provided by the reconfiguration algorithm. The research questions will thereby be as follows.

- Does the implementation of the self-stabilizing reconfiguration algorithm provide a correct execution?
- How does the efficiency and performance of the self-stabilization part of the reconfiguration algorithm, i.e the bootstrapping when a fault has occurred, scale as the number of processors changes?
- Does the quorums from the self-stabilizing reconfiguration algorithm provide the shared memory emulation algorithm with usable and correct quorums?
- What will the performance of the reconfiguration algorithm be?
- What will the performance of the shared memory emulation be?
- What will the performance of the shared memory emulation be when paired with the reconfiguration algorithm?

The correctness of the reconfiguration algorithm is evaluated based on invariants from the paper in which the algorithm was originally presented. The performance is evaluated in terms of latency, the number of messages sent and delivered, as well as the number of rounds needed to complete a reconfiguration. The bootstrapping is measured from the start in a faulty state to when the algorithm returns to a safe state. The evaluation of the shared memory emulation performance is done by measuring latency as well as the number of messages sent and delivered needed from

when a value is initially written by a processor until the point when it is available for the other processors.

6.2 Evaluation Criteria

The system needs to live up to a set of criteria in terms of correctness and performance. When it comes to correctness, the system needs to pass a set of invariants. How the performance of the reconfiguration algorithm is affected when the number of processors increases is examined to see if it is feasible to use in a real-world setting, in terms of latency, full loop iterations, and messages sent and delivered. The reconfiguration algorithm must also be able to provide quorums that can be used by a shared memory emulation algorithm. In addition to working as an evaluation tool for the reconfiguration algorithm, the shared memory emulation must also provide good performance when the number of processors increases. Since to the best of our knowledge, no other similar system has been examined under similar conditions, no comparison can be made between the measured results in this project and other projects. It is therefore only possible to examine how it scales and not the exact latency or the number of messages communicated.

6.3 Experiment Description

The conducted experiments used for the evaluation are the following:

6.3.1 Validate the Correctness of the Reconfiguration Algorithm

The correctness is validated based on invariants and lemmas. The invariants and lemmas can be seen in the appendix 8.5 and derived from the paper [1]. They have then been combined into a debugger tool which is executed in parallel with the reconfiguration algorithm, thereby allowing local run time validation of the reconfiguration algorithm.

6.3.2 Establish the Baseline Performance of the Shared Memory Emulation

The baseline performance of the shared memory emulation in isolation will be evaluated in terms of messages sent/delivered, number of communication and latency, per read and writer operation. The measured performance is measured from the perspective of the processor which initiates the read/write call and finishes when the processor receives confirmation that the operation was successful. This is done for processor configurations between three and nine processors and the number of processors available in the system is the number of processors agreeing on a read/write operation. The experiment is conducted continuously for 180 seconds and the average metric across the operations is what is reported. The experiment is conducted to have the baseline performance of the shared memory emulation algorithm and

thereby be able to compare the baseline results to the results when the shared memory emulation is paired with the reconfiguration algorithm.

6.3.3 Establish the Baseline Performance of the Reconfiguration Algorithm

The baseline performance is measured in isolation from the application. The baseline will be used to evaluate the results from the later experiment where the reconfiguration algorithm is paired with an application. The performance of the join, bootstrap, and reconfiguration will be measured in terms of the number of messages sent/delivered, number of iterations, and latency for each operation. The join mechanism will be measured from the perspective of the joining processor. The measurements start when an initial request to join is sent and ends when the joining processor has become a participant (note that the joining processor is not part of the configuration and would require a full reconfiguration to become a member). When conducting the joining experiment only a single processor is joining regardless of the processor configuration. This is done to evaluate how the performance is affected by a single processor joining with different number of processors having to accept the joining processor (note that the single joining processor is counted as part of the system i.e evaluating nine processors means that eight have to accept the one joining processor). The bootstrap performance is measured from the point at which a single processor is put in a faulty state to the point where it has returned to a safe and correct state. As with the joining mechanism only a single processor is put in a faulty state regardless of the number of processors in the system and the performance of a single faulty processor in a system with different processor configurations is measured. The reconfiguration mechanism is measured from the point at which a new configuration is proposed to when a new quorum has been established. The new quorum is proposed by a single processor and the number of processors in the system is the number of processors that have to agree on the new quorum. The join, bootstrap, and reconfiguration experiments are all conducted continuously over 180 seconds and the average from all measured operations is reported and used in the evaluation.

6.3.4 Establish the Performance of the Shared Memory Emulation and Reconfiguration when Paired

This experiment aims to evaluate the performance of the shared memory emulation and reconfiguration algorithm when paired together. The results from the tests when paired will then be compared with the results from the isolated experiments in order to find the performance impact the algorithms have when paired. The experiment will combine the read/write operation from the shared memory with the join and reconfiguration operations from the reconfiguration algorithm. The metrics will be measured in the same manner as when evaluated in isolation to provide comparable results. A note should be made that when pairing read/write operation with a reconfiguration, a quorum consisting of the old quorum in union with the new will be used in order to provide the shared memory emulation with a functioning quorum

even when a reconfiguration is undergoing.

6.3.5 Evaluating Correctness in a Failure Prone Network

In order to evaluate the correctness when subjecting the processors to periodic as well as delays in the start-up phase, an experiment is conducted. The experiment is designed to evaluate how the processors handle network delays. What is done is that when the processors are started, they all start with a delay, thus imitating a network where messages are sent but no response is immediately given. This ensures that the system can function properly in a non-ideal network and thus exhibit a self-stabilizing behavior.

6.3.6 Number of Conducted Experiments

Running the experiments continuously for 180 seconds means that the number of times each experiment is conducted varies depending on the time it takes to finish an operation. Thus the actual number of times the experiment is evaluated can be calculated by dividing the total time of 180 seconds by the average time for a single operation to finish. This allows us to maximize the number of times each experiment is conducted however, at the expense of running each experiment the exact same number of times.

6.4 Evaluation Environment

The system is evaluated on a service known as SSC (Swedish Science Cloud) provided by SNIC (Swedish National Infrastructure for Computing). SSC allows for the creation of multiple instances (virtual computers) running its own operating system in a fully connected network. SSC uses OpenStack as its way of handling the management and creation of the instances, this provided a good, clean environment in which the evaluation could be done. When evaluating the system, an executable file was uploaded to all instances and executed independently at the same time. The communication between the instances when executing the code was all done via an internal network where only a single member of the network had the ability to communicate with machines outside the internal network. The network topology of the network used can be seen in figure 6.1. All instances used in the evaluation had a single virtual CPU with 512 Mb of RAM, a 20Gb hard drive and ran the Linux distribution Ubuntu 18.04 as its OS.

The software used was a Rust implementation of the algorithms executing in a single loop where messages were sent and received once in each loop iteration. The implementation consisted of approximately 3000 rows of code and within these 3000 lines, all algorithms used were implemented. The same code ran on all processors and was uploaded as a pre-compiled executable file, thus allowing the processors to save resources by avoiding the task of compilation.

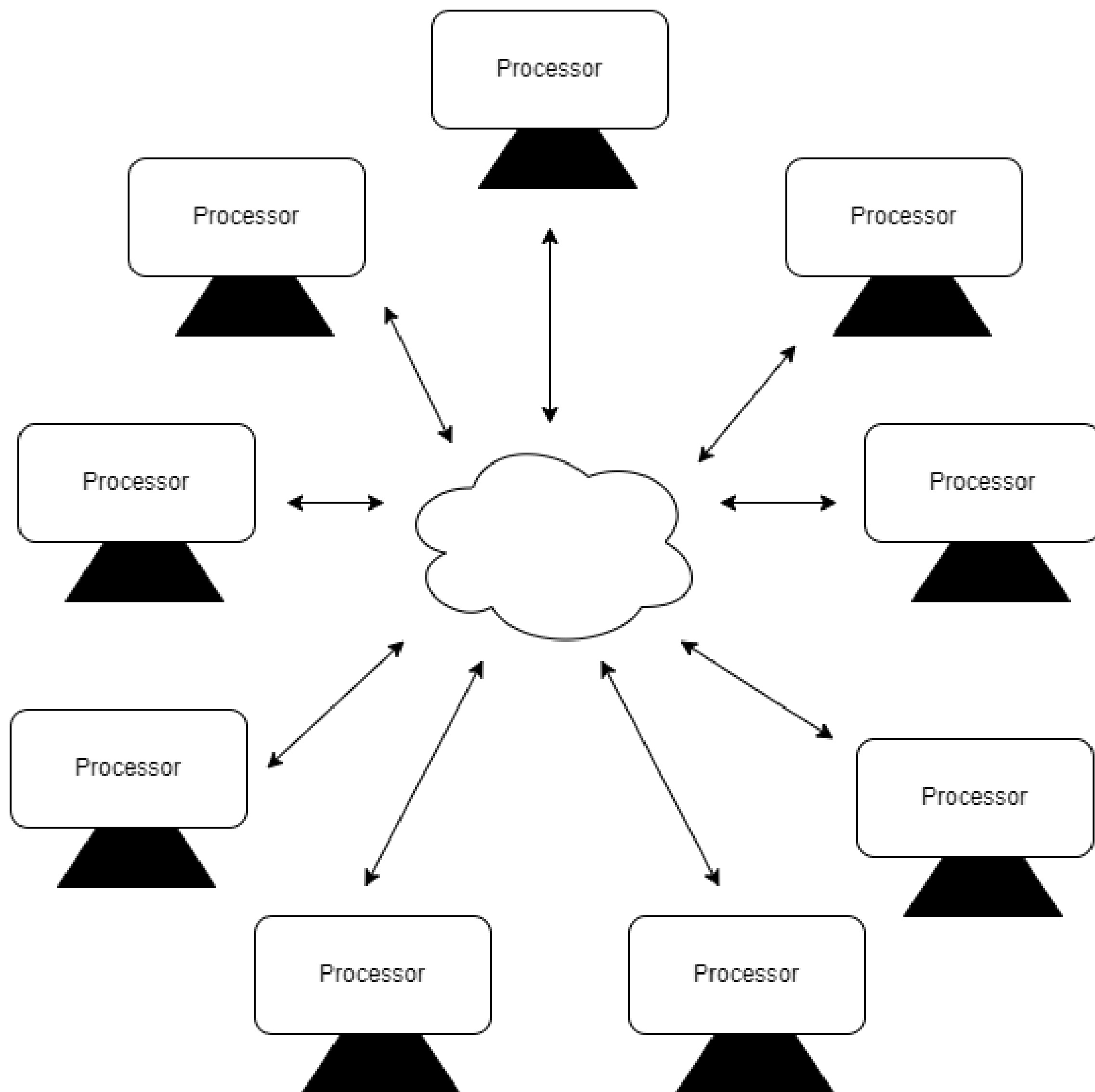

Figure 6.1: The network topology on which the implementation was evaluated

7 Evaluation Results

We are interested in evaluating the performance of reconfiguration functionality in the context of the studied application, which is multi-reader/multi-writer (MRMW) atomic shared memory emulation. As a baseline evaluation, we establish the performance of the reconfiguration functionality in isolation from its application (Section 7.1) as well as the performance of the application in isolation from reconfiguration (Section 7.2). Then, we focus on studying the behavior of the application when it is running concurrently with reconfiguration functionality, such as join and reconfiguration (Section 7.3).

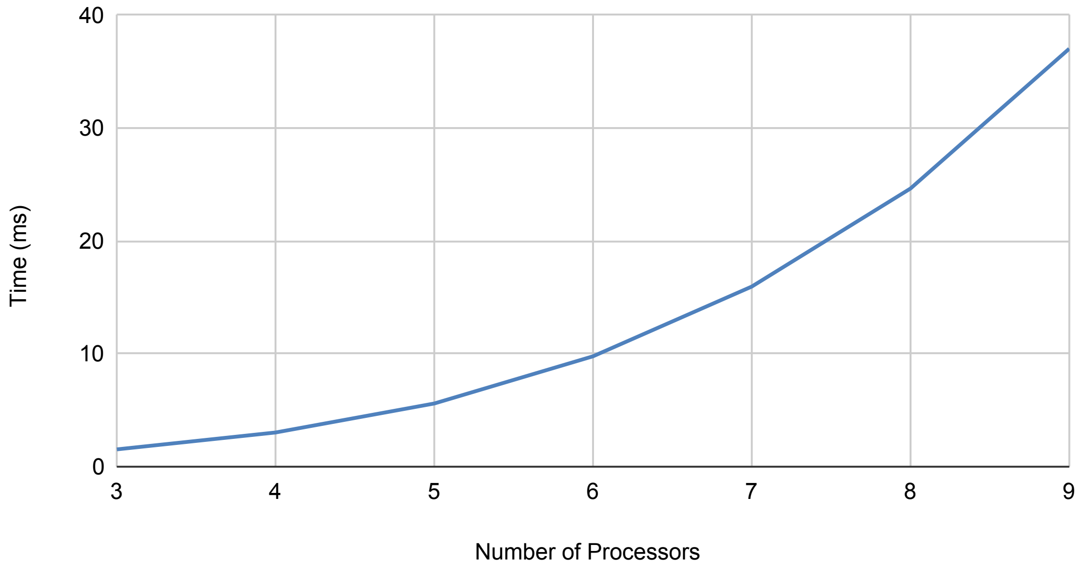

Figure 7.1: The average time it takes to finish a single iteration

7.1 Evaluating the Self-stabilizing Reconfiguration in Isolation from the Application

As previously mentioned and described in **??**, we establish the baseline performance of the reconfiguration functionally, by studying the reconfiguration and join operation (that run in isolation of the application). We are also interested in cases in which the reconfiguration service has to bootstrap itself, say when starting to run or after the occurrence of a transient fault. Our goal is to understand the latency of these operations. We note that latency is mainly influenced by the number of full loop iterations as well as the number of messages send and delivered. Therefore, we start our investigation by looking at these numbers.

7.1.1 Number of Iterations

By examining the code, we see that join and reconfiguration operations cannot finish before they complete 6, and respectively, 2 iterations. The reason is that the reconfiguration works by having coordinated steps not only through the three phases but also through degrees. There are six degrees and these are used to make sure that no one changes phase before it has been noticed by all other processors and they can move to the next phase in a coordinated manner. Also, during bootstrapping periods, at least 2 full loop iterations are needed because there needs to be at least one round in which the empty-configuration symbol propagates and then another round for disseminating the value of the new configuration.

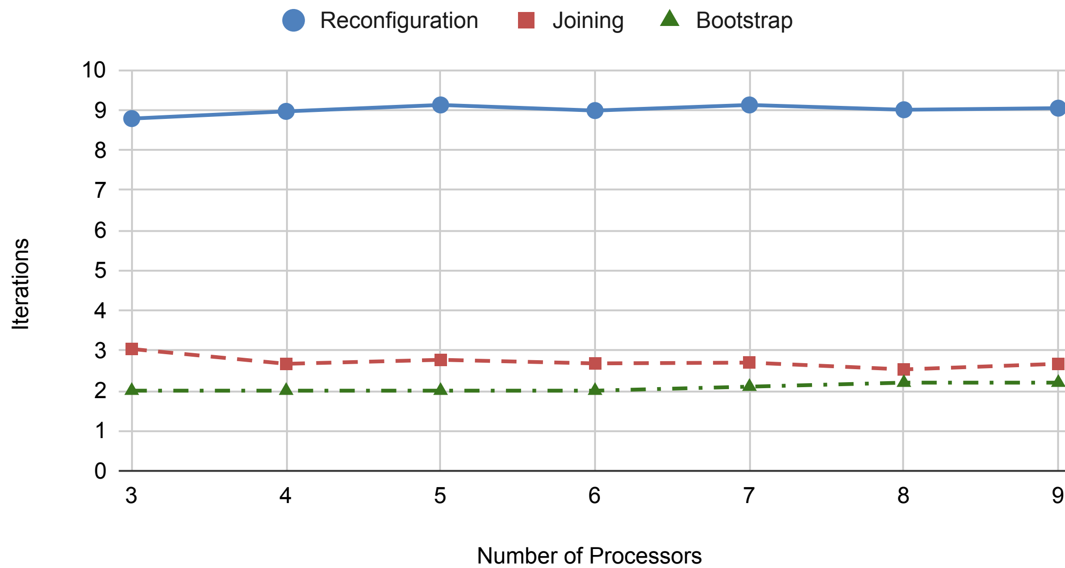

Figure 7.2: The average number of complete loop iterations required to carry out different reconfiguration operations

Our expectation based on the code indicated that the reconfiguration and join would need six and two iterations to finish, however as can be seen in 7.2 the results show a need for more iterations than predicted. The reason behind this is due to the implementation of the reconfiguration algorithm which is done in a way that only allows for messages to be delivered in every iteration. Therefore, the extra iterations necessary is believed to stem from the asynchrony of the system which means that there is no guarantee when a message from another processor is delivered. This means that some extra iterations will be done where no changes will be made, waiting for the other processors to answer. As expected two full loop iterations were required for the bootstrap operation. Furthermore, the evaluation also indicates that the number of processors does not affect the number of iterations needed.

7.1.2 Number of Messages Sent and Delivered

Join and reconfiguration operations as well as bootstrapping exchange messages via broadcast rounds, i.e., $O(n)$ messages in every iteration and $O(n^2)$ in every communication round. Thus, in an ideal setting, the number of messages sent and delivered are expected to be equal and should correspond with the number of iterations and full loop iterations executed. In exact terms, the expected number of messages sent and delivered should be the total number of processors in the system, times the number of iterations.

Figure 7.3, shows that, as expected, the number of messages sent and delivered does correspond well to the number of iterations necessary to complete the opera-

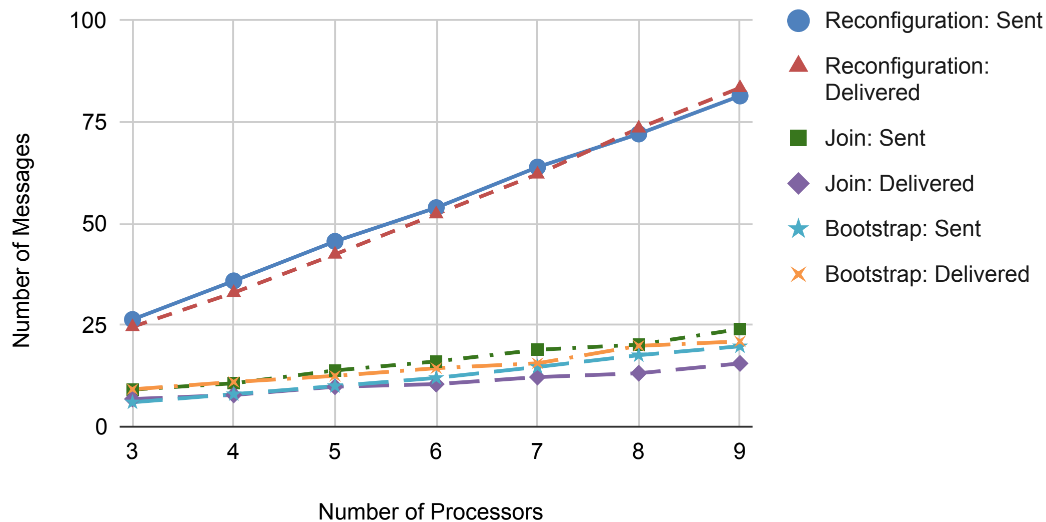

Figure 7.3: Average number of messages sent and delivered during different reconfiguration operations

tion. However, before conducting the experiments, it was believed that the number of messages sent and delivered would be equal, which was not the case. An explanation for this as mentioned before, messages are both broadcasted and received in every iteration. This means that if a processor manages to complete more rounds than another processor, it will also broadcast more messages and vice versa. This is believed to be the cause of the number of messages sent and delivered is different.

7.1.3 Operation Latency

The operation latency is expected to depend on the number of iterations necessary, which are six, two, and two for the reconfiguration, join, and bootstrap respectively. In addition, we expect that the processing time will increase as the number of processors in the system increases and with it the latency as well. It is therefore expected that the latency for all operations will increase but the more iterations necessary to complete the operation, the more the latency will be affected when the number of processors increases. Furthermore, the predicted latency based on the average latency for a single iteration (7.1) multiplied by the number of complete loop iterations (7.2) for the different parts of the reconfiguration can be seen in figure 7.4.

As expected the latency for an operation grows as the number of processors increases which can be seen in 7.4. Furthermore, it is also clear that the latency is heavily influenced by how many iterations it takes to finish the operation. This explains why the reconfiguration, which takes nine iterations to finish, has a much steeper increase in latency compared to the join and bootstrap, which only need

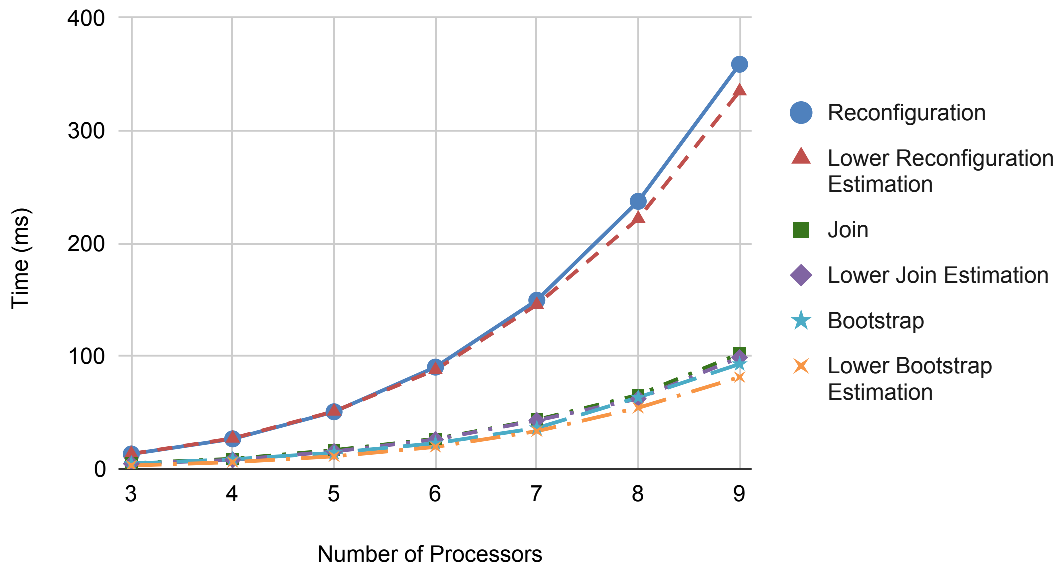

Figure 7.4: Average latency of performing different reconfiguration operations

around three iterations to finish. When comparing the estimated latency with the measured, we note that the bootstrap estimation is almost identical to the measured results. We also note that both the reconfiguration as well as join estimation are close to the measured result, with the largest difference in the reconfiguration being 7% and for the join 4%.

7.2 Evaluating the Shared Memory Emulation in Isolation from the Self-stabilizing reconfiguration

By conducting the experiment in 6.3.2A baseline was established for the shared memory emulation when running in isolation and the results will be used to compare with the results when it is later paired with the reconfiguration. We want to evaluate the performance in terms of the number of iterations as well as the number of messages sent and delivered, for both read and write operations. The reason for this is that similarly to the self-stabilizing reconfiguration algorithm we believe that these two metrics can be used to explain the latency of the read and write operations.

7.2.1 Number of Iterations

When analyzing the implementation, we expect both a read and a write operation to take seven iterations to complete. This is independent of how many processors are part of the system. The reason behind this comes down to how the algorithm

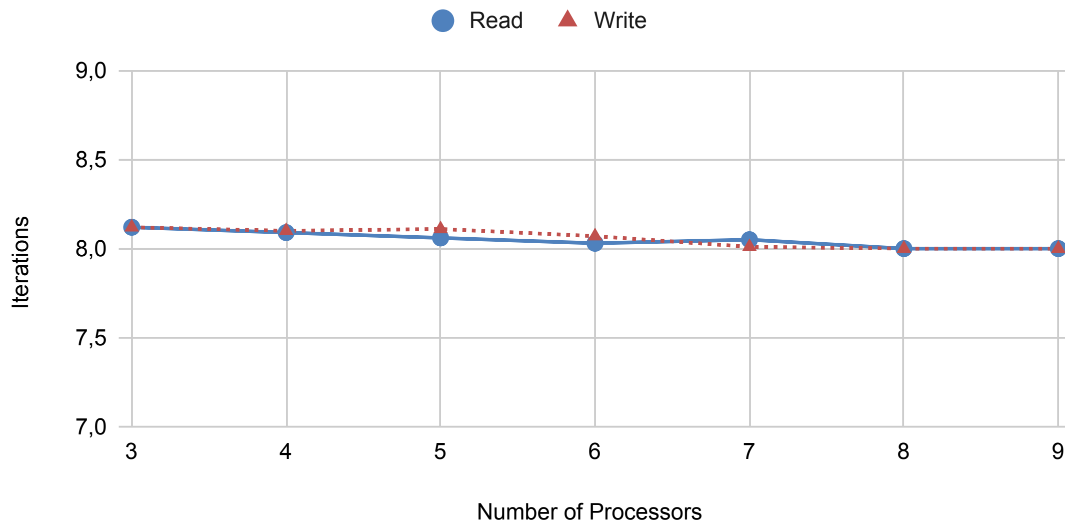

Figure 7.5: Average number of complete loop iterations required for different emulation operations

was implemented, of which a description can be found in chapter 4.

As mentioned it was expected that a read and write operation would take seven rounds, however as can be seen in 7.5 it takes about eight iterations to finish both a read and write operation. That it takes the same number of iterations for both reads and writes was expected and the extra iteration necessary to complete the operation is believed to be for the same reason as in the case of the reconfiguration algorithm. Namely, the fact that it takes additional iterations where the processor needs to wait for incoming answers from the other processors in the system.

7.2.2 Number of messages

The number of messages delivered and sent is expected to be the same for both reads and write operations with respect to a specific number of processors. The exact number of messages necessary to be sent or broadcasted was expected to be two times the number of processors you want to access and thus the message complexity is expected to be $O(n)$.

The number of messages sent and delivered can be seen in 7.6 and confirms the expectations, both in terms of the number of messages sent and delivered as well as the fact that the same number of messages were sent as delivered.

Figure 7.6: Average number of messages sent and delivered during different emulation operations

7.2.3 Operation Latency

Since the implementation is done in a loop, the number of iterations necessary to complete the operation is the main factor for the latency, similar to the reconfiguration. Like the reconfiguration, it is expected that the time it takes to complete one iteration grows as the number of processors in the system increases, due to the increased data processing necessary. We therefore, expect that the latency will be affected similarly to how the reconfiguration operation latency increases. Furthermore, the estimated lower latency for the read and write operations based on the number of complete loop iterations (7.5) multiplied with the average time for a single complete iteration (7.1) can be seen in 7.7. We note a difference in latency between the estimated and measured latency for the read and write operation, of at most 6%, 9.3% respectively.

The measured latency can be seen in 7.7 and as expected it grows as the number of processors increases. The increase in latency is due to the iterations taking more time when having to process more data when the number of processors increases, a hypothesis which can be supported further as more data processing has to be done as a direct consequence of the increase in messages sent and delivered.

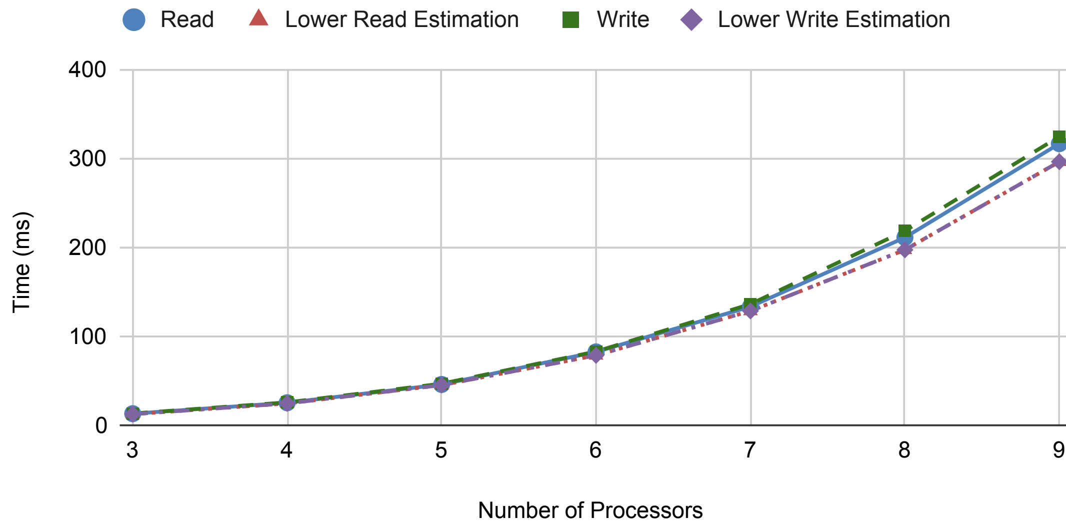

Figure 7.7: Average latency of performing different emulation operations

Figure 7.8: Average latency of performing different emulation operations with a constant number of writers and a varying number of readers

Figure 7.9: Average latency of performing different emulation operations with a constant number of readers and a varying number of writers

7.3 Evaluating the shared memory emulation paired with the self-stabilizing reconfiguration

We want to compare the isolated performance of the self-stabilizing reconfiguration algorithm and the shared memory algorithm when paired together. This will be done by pairing the join and reconfiguration operations with the read and write operations from the shared memory emulation algorithm, as described in 6.3.4. The goal of doing this is to evaluate how the performance is affected by the individual operations being paired together. In order to be able to compare the operations, the operations will be evaluated in the same manner as well as on the same metrics as when they were evaluated in isolation.

7.3.1 Number of Iterations

We expect that the number of iterations necessary to complete an operation will be similar to what the results from the baselines are. This is due to the operations being implemented in a way that allows for them to execute without having to wait for the operation which it is paired with. The join will for instance not affect the number of iterations the reconfiguration operation needs and the read and write operation can be executed even though a reconfiguration is currently taking place.

As can be seen in 7.10, the number of iterations stays close to what they were in the isolated experiments, which was predicted for all operations.

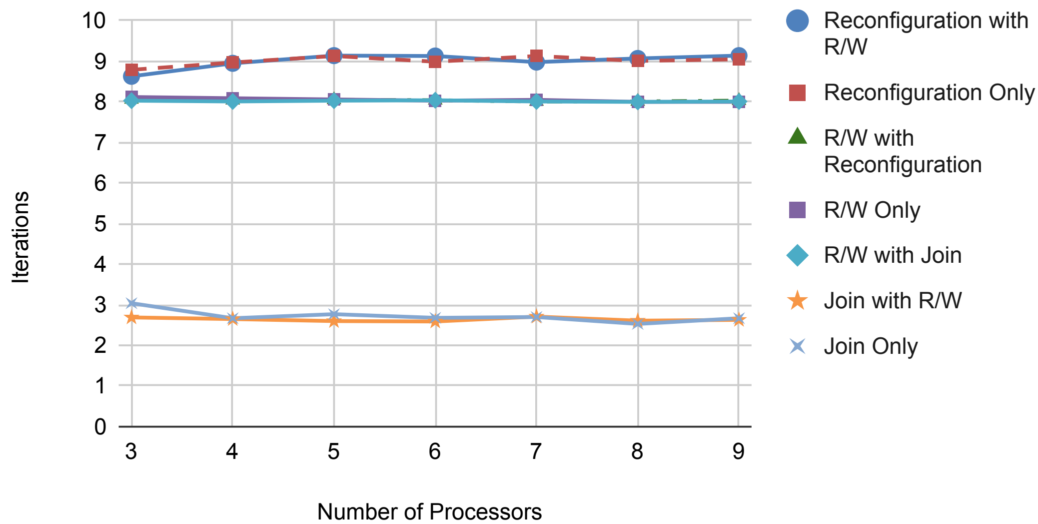

Figure 7.10: Average number of complete loop iterations required to perform emulation- and reconfiguration operations concurrently

7.3.2 Number of Messages

The number of messages necessary to complete an operation is also expected to be similar to what is necessary when executing in isolation. This is because as for the number of iterations, the operations are expected to be able to execute without having any interference from the operation it is paired with and thus the number of messages both sent and delivered should stay the same as for the isolated case.

As predicted the messages necessary do not seem to be affected by pairing operations, instead they show the same behavior as when executed in isolation. This can be seen in 7.11 and 7.12.

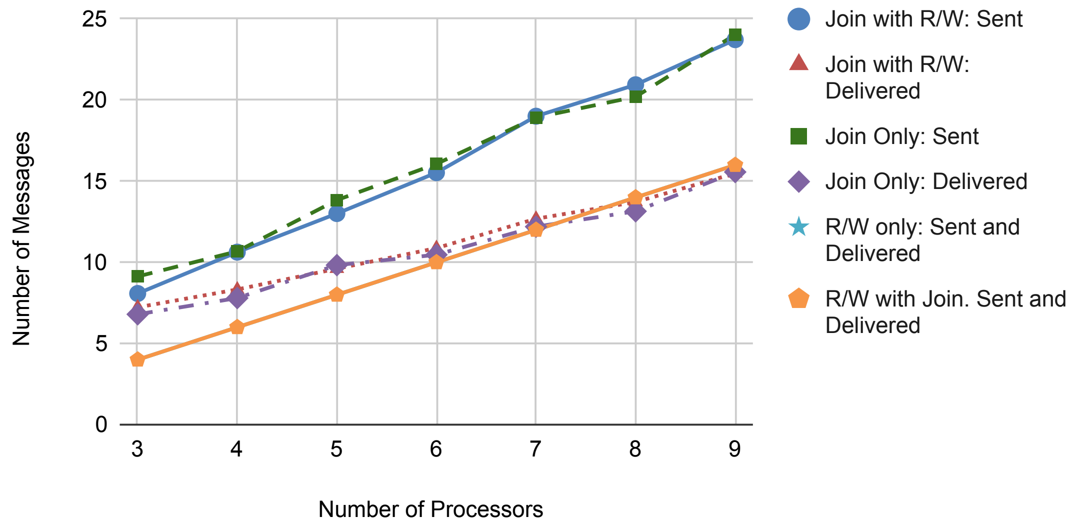

Figure 7.11: Average number of messages sent and delivered when performing emulation- and join operations concurrently

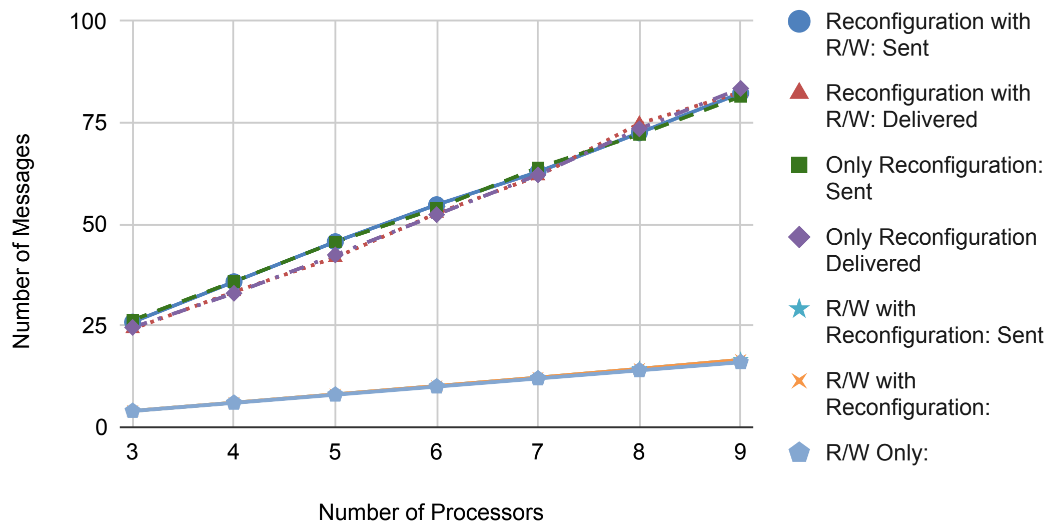

Figure 7.12: Average number of messages sent and delivered when performing emulation- and quorum reconfiguration operations concurrently

7.3.3 Operation Latency

Even though we do not expect to see an increase in iterations nor messages sent and delivered compared to when the operations ran isolation, we do expect to see an increase in latency. The reason to why this is expected is similar to what it assumed to grow the latency as the number of processors increases in the isolated cases, namely additional data processing. Having two operations simultaneously is expected to require more processing and thus the time it takes to finish an iteration increases. This in turn means that when combined it is expected that the latency will grow more as the number of processors increases compared to when the operations were executed in isolation.

In 7.13, we can see that our prediction is confirmed. We see that when two operations are paired, it results in additional latency compared to the latency of the isolated operation. However, the increase in latency is very small with the largest difference being an increased latency of 4.1% when comparing reconfiguration with shared memory emulation to only reconfiguration. The latency comparison between R/W with reconfiguration and only R/W, we see that the largest difference is an increase in latency of 1.35%.

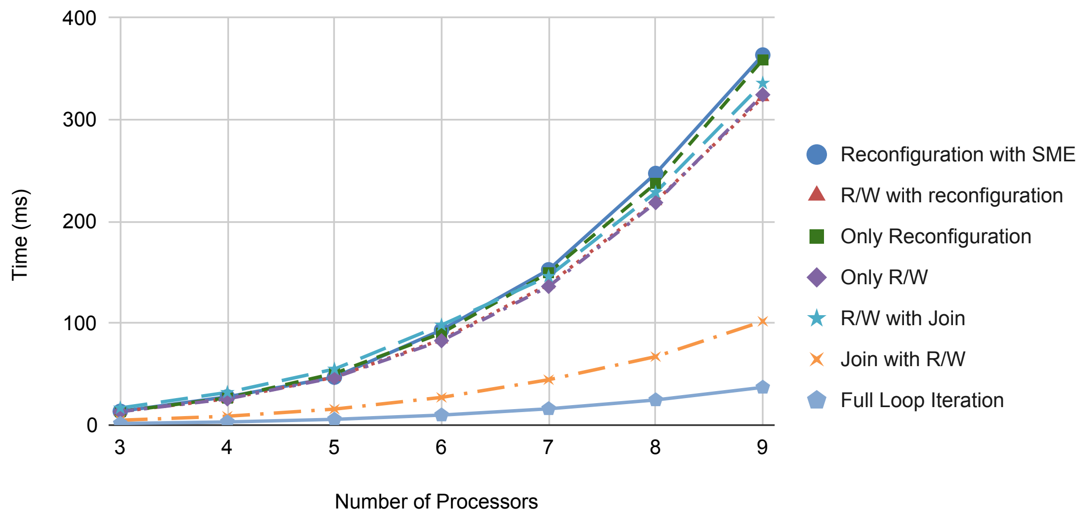

Figure 7.13: Average latency of performing different emulation- and reconfiguration operations concurrently

7.4 Validation

The self-stabilizing quorum reconfiguration was validated through the use of a debugger consisting of lemmas and invariants taken from the paper its original paper by DGMS [1]. Due to limitations of using the snapshot algorithm in this particular setting, validation was only possible to perform on a local level and thus no guarantees of correctness on a global level can be made. However, based on the results from the debugger we can say that on a local level our implementation showed no sign of faulty behavior. This means that to the best of our knowledge and validation, our implementation was correct but it is important to note that this was only a limited validation and did not have the ability to capture all faulty behavior. Furthermore, the MRMW atomic shared memory emulation by Schvartsman and Lynch was able to use the quorums it was given by the DGMS reconfiguration algorithm and showed the desired behavior. Thus, the quorums provided will be regarded as correct.

The experiment in which delays were used to simulate a network where messages are not answered (6.3.5), showed no sign of faulty behavior thus indicating that the algorithm, as well as its implementation of it, was correct. This further proves that the quorum reconfiguration algorithm works as intended and is able to handle these types of faults.

8

Discussion & Conclusion

In this section, a discussion regarding the project and potential future work is presented. Lastly, a conclusion of the project is presented.

8.1 Snapshot Algorithm and Global Validation

Some invariants used to ensure the correct functionality of the implementation of the global reset reconfiguration algorithm requires, as previously mentioned, the ability to gain a global view of the state of each participating processor. This ability must furthermore be available at any instance in time. Due to the asynchrony of the state transitions, this is not possible. Although, by making use of a snapshot mechanism on top of the implementation, it is possible to *sometimes* gain a global view of the system at an instance in time. To be able to produce a snapshot, however, requires that every processor does not make any changes to any of their local variables during a specific time-frame. This time-frame is the time it takes for the initiator of the snapshot to send and receive two messages from every processor, namely 2 RTT_{max}, where RTT_{max} is the largest round trip time (RTT) of any message-channel used by the initiator. If any processor makes any changes to its local variables during this time-frame, the snapshot needs to be re-initiated.

During the implementation of the snapshot algorithm, one decision that was taken was to make use of piggybacking. All of the messages that it produced were attached to the messages sent related to the reconfiguration algorithm. The reasoning behind this decision was that it thereby would not have as great of an impact on the performance of the reconfiguration algorithm, due to the fact that no additional messages would need to be dealt with. However, by not producing any additional messages, the chances of successfully completing the snapshot mechanism were reduced. Since the reception of a message that indicates a state transition by another processor often triggers the receiver to change its own state, having snapshot-related messages piggybacked makes it difficult to create a snapshot during a delicate replacement.

During the initial evaluation of the implementation, the described difficulty of producing successful snapshots became eminent, as all the snapshots either occurred before or after the delicate replacement took place. One way of partly circumventing this problem was to not act on the values received from other processors every round, but instead every fifth or tenth round. This change made it possible to increase the rate of successful snapshots during an active reconfiguration, and hence also ensure

correct functionality via the invariants. However, even with these changes, the snapshot mechanism is still seldom successful during phase transitions, making it hard to effectively check the global invariants. Furthermore, these changes to the reconfiguration algorithm drastically worsened its performance, making it impossible to use these configurations in a real-world setting. Hence, as a future improvement, it is possible to consider two different modes of implementation, one which emphasizes performance and the other one used for validation.

8.2 Validation Limitations

When developing the debugger for the validation we attempted to cover as many different invariants and lemmas as we could within the scope of this project. This means that although the debugger can detect many of the possible faulty behaviors, it does not cover all of them. Because of this, it is not possible for us to prove a completely bug-free implementation of the self-stabilizing quorum reconfiguration algorithm by DGMS. It would therefore be a possibility for future work to further extend our debugger with more lemmas and invariants, both on a local as well as a global level. Having a more advanced debugger would also give our implementation and results in greater legitimacy since it could lead to the discovery of faulty behavior or lack thereof.

In addition to improvements in the debugger for the reconfiguration algorithm by DGMS, a debugger for the validation of the Schvartsman and Lynch MRMW atomic shared memory emulation implementation could be a good addition. This would not only lead to the validation of the implementation of the Schvartsman and Lynch algorithm but also as further proof of a correct implementation of the DGMS reconfiguration algorithm. This is since the Lynch and Schvartsman algorithm relies on quorum, which in our case comes from the DGMS reconfiguration. Therefore, if a debugger was developed which could prove the correct behavior of the Schvartsman and Lynch implementation it could detect incorrect behavior which could stem from being provided faulty quorums.

8.3 Implementation and Evaluation Limitations

When implementing the different algorithms that made up the system, the papers in which the respective algorithm was originally published were used. All of the papers did provide pseudo-code which was very helpful when implementing. Once implemented, all algorithms were validated in order to make sure of correct behavior. The implementation and algorithm validated to the largest extent were the reconfiguration algorithm by DGMS which was validated using a number of invariants. These invariants were based on the paper and the proofs available and used to validate throughout the execution locally. Validating the execution against the execution meant that, we could discover potential bugs and inconsistencies in our implementation or the algorithm itself. When a correct behavior was validated, performance tests were made on the SNIC network which gave the ability to test the

performance and validate the correctness in a distributed setting. Something that should be noted with the SNIC SSC network is that you are given virtual machines. However, there is to the best of our knowledge, no way of knowing how these virtual machines were located, i.e if they ran on the same computer or were physically separated. Furthermore, when evaluating on SNIC SSC, we could sometimes notice small differences in performance depending on the day we ran the evaluation. Furthermore, when evaluating the implementation we were only able to do so with a maximum of nine processors in the system.

8.4 Takeaways and Future Work

Our results show that for a system consisting of three up to nine processors, having a self-stabilizing quorum reconfiguration on top of an atomic shared memory emulation does not have a large impact on performance and has the ability to provide the system with its intended functionality. We hope these findings can be used as a motivation to use fault tolerant, self-stabilizing quorum consensus algorithms in other projects with a similar setting. Due to the increased motivation, we hope that this can lead to more projects in the future making use of these algorithms and thereby also having more robust systems in the future. Furthermore, having a project like this that shows the possibilities and feasibility of self-stabilization can hopefully lead to more projects researching self-stabilization for other algorithms and in different system settings, thereby moving the field forward. Looking at this project and the reconfiguration algorithm by DGMS more specifically, we show that this advanced algorithm can be implemented and made to function (to the best of our knowledge) in a correct and intended way. This project can therefore be seen as a practical validation and evaluation of an algorithm that previously had only been theoretically proven. This gives the algorithm an even greater status and could also lead to adoption in future research and industry projects.

8.5 Conclusion

The results indicate that our implementation of the shared memory emulation on top of quorum reconfiguration validates that the quorum reconfiguration algorithm by DMSG, to the best of our knowledge, appears to offer correct behavior and satisfies the intended functionality. Furthermore, based on the evaluation we see that the use of reconfiguration paired with shared memory emulation leads to no significant loss in performance, compared to when evaluated individually. This indicates that it is possible to have a fault tolerant self-stabilizing reconfiguration without losing performance when it is paired with a MRMW atomic shared memory algorithm.

This report presents an in-depth explanation and evaluation of the individual algorithms as well as the paired evaluation. The explanations brake down the algorithms in a way more oriented toward how to implement the algorithms in practice. Thus, it can be seen as both a guide to how they were implemented in this project and as a complement to their original papers. In addition to providing evaluation results, a proposed explanation to the results is given which indicates that the number of complete loop iterations stays fairly constant and that the main factor when it comes to latency is the time it takes to finish one loop which increases as the number of processors in the system grows. Moreover, the report shows how an advanced algorithm can be implemented in a readily accessible way through a single loop divided into different phases without the addition of multiple threads. The implementation is also validated by a debugger consisting of invariants and lemmas which further proves the ability to take an advanced algorithm and make a more accessible version while still having correct behavior.

The results of this project can be of use when implementing or planning to implement a similar project in the future. Furthermore, we hope that this project shows the potential of using self-stabilizing algorithms and that having a robust fault tolerant version of an algorithm does not have to result in a performance loss. Finally, we hope that this can be used as a foundation for future projects in which algorithms can be both improved as well as extended for a more mature implementation, evaluation, and validation.

Bibliography

[1] Shlomi Dolev et al. "Self-stabilizing Reconfiguration". In: Lecture Notes in Computer Science 10299 (2017), pp. 51–68. DOI: 10.1007/978-3-319-59647-1_5. URL: https://doi.org/10.1007/978-3-319-59647-1%5C_5.

[2] N.A. Lynch and A.A. Shvartsman. "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts". In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 1997, pp. 272–281. DOI: 10.1109/FTCS.1997.614100.

[3] Maarten van Steen and Andrew S. Tanenbaum. Third. 2020, p. 7. URL: https://www.distributed-systems.net/index.php/books/ds3/ds3-ebook/.

[4] Edsger W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control". In: *Commun. ACM* 17.11 (Nov. 1974), pp. 643–644. ISSN: 0001-0782. DOI: 10.1145/361179.361202. URL: https://doi.org/10.1145/361179.361202.

[5] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN: 0-262-04178-2. URL: http://www.cs.bgu.ac.il/%5C%7Edolev/book/book.html.

[6] Nancy Lynch and Alexander Shvartsman. "RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks". In: Oct. 2002, pp. 173–190. ISBN: 978-3-540-00073-0. DOI: 10.1007/3-540-36108-1_12.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing Memory Robustly in Message-Passing Systems". In: *J. ACM* 42.1 (Jan. 1995), pp. 124–142. ISSN: 0004-5411. DOI: 10.1145/200836.200869. URL: https://doi.org/10.1145/200836.200869.

[8] Carole Delporte-Gallet et al. "Implementing Snapshot Objects on Top of Crash-Prone Asynchronous Message-Passing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 2033–2045. DOI: 10.1109/TPDS.2018.2809551.

[9] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. *Self-Stabilizing Snapshot Objects for Asynchronous Fail-Prone Network Systems*. 2019. DOI: 10.48550/ARXIV.1906.06420. URL: https://arxiv.org/abs/1906.06420.

[10] Marco Canini et al. "Renaissance: A self-stabilizing distributed SDN control plane using in-band communications". In: *J. Comput. Syst. Sci.* 127 (2022), pp. 91–121.

[11] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-stabilizing Multivalued Consensus in Asynchronous Crash-prone Systems". In: *EDCC*. IEEE, 2021, pp. 111–118.

[12] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-Stabilizing Indulgent Zero-degrading Binary Consensus". In: *ICDCN*. ACM, 2021, pp. 106–115.

[13] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-Stabilizing Set-Constrained Delivery Broadcast (extended abstract)". In: *ICDCS*. IEEE, 2020, pp. 617–627.

[14] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-stabilizing Uniform Reliable Broadcast". In: *NETYS*. Vol. 12129. Lecture Notes in Computer Science. Springer, 2020, pp. 296–313.

[15] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. "Self-stabilizing Byzantine-tolerant Broadcast". In: *CoRR* abs/2201.12880 (2022).

[16] Chryssis Georgiou et al. "Loosely-self-stabilizing Byzantine-Tolerant Binary Consensus for Signature-Free Message-Passing Systems". In: *NETYS*. Vol. 12754. Lecture Notes in Computer Science. Springer, 2021, pp. 36–53.

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

[18] Shlomi Dolev et al. "Self-stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors". In: *CSCML*. Vol. 10879. Lecture Notes in Computer Science. Springer, 2018, pp. 84–100.

[19] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. "Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery - (Extended Abstract)". In: *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*. Ed. by Vincent Gramoli and Rachid Guerraoui. Vol. 7853. Lecture Notes in Computer Science. Springer, 2013, pp. 42–57. DOI: 10.1007/978-3-642-40148-0_4. URL: https://doi.org/10.1007/978-3-642-40148-0%5C_4.

[20] Shlomi Dolev and Elad Schiller. "Self-stabilizing group communication in directed networks". In: *Acta Informatica* 40.9 (2004), pp. 609–636. DOI: 10.1007/s00236-004-0143-1. URL: https://doi.org/10.1007/s00236-004-0143-1.

[21] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. "Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks". In: *IEEE Trans. Mob. Comput.* 5.7 (2006), pp. 893–905. DOI: 10.1109/TMC.2006.104. URL: https://doi.org/10.1109/TMC.2006.104.

[22] Joffroy Beauquier, Thomas Hérault, and Elad Schiller. "Easy Stabilization with an Agent". In: *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*. Ed. by Ajoy Kumar Datta and Ted Herman. Vol. 2194. Lecture Notes in Computer Science. Springer, 2001, pp. 35–50. DOI: 10.1007/3-540-45438-1_3. URL: https://doi.org/10.1007/3-540-45438-1%5C_3.

[23] Shlomi Dolev et al. "Practically-self-stabilizing virtual synchrony". In: *J. Comput. Syst. Sci.* 96 (2018), pp. 50–73. DOI: 10.1016/j.jcss.2018.04.003. URL: https://doi.org/10.1016/j.jcss.2018.04.003.

[24] Zacharias Georgiou et al. "A Self-stabilizing Control Plane for Fog Ecosystems". In: *13th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2020, Leicester, United Kingdom, December 7-10, 2020*. IEEE, 2020, pp. 13–22. DOI: 10.1109/UCC48980.2020.00021. URL: https://doi.org/10.1109/UCC48980.2020.00021.

[25] Marco Canini et al. “A Self-Organizing Distributed and In-Band SDN Control Plane”. In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. Ed. by Kisung Lee and Ling Liu. IEEE Computer Society, 2017, pp. 2656–2657. DOI: 10.1109/ICDCS.2017.328. URL: https://doi.org/10.1109/ICDCS.2017.328.

[26] António Casimiro, Emelie Ekenstedt, and Elad Michael Schiller. “Self-Stabilizing Manoeuvre Negotiation: The Case of Virtual Traffic Lights”. In: *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 354–356. DOI: 10.1109/SRDS47363.2019.00048. URL: https://doi.org/10.1109/SRDS47363.2019.00048.

[27] J. Postel. *RFC0768: User Datagram Protocol*. USA, 1980.

[28] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. “Sharing Memory Robustly in Message-Passing Systems”. In: *J. ACM* 42.1 (Jan. 1995), pp. 124–142. ISSN: 0004-5411. DOI: 10.1145/200836.200869. URL: https://doi.org/10.1145/200836.200869.

A

Appendix 1

A.1 Lemmas and Invariants

All invariants and lemmas are taken or derived from the paper, *Self-Stabilizing Reconfiguration* [1] by Dolev, Georgiou, Marcoullis and Schiller.

Invariant 1

$phs_i[k] = phs_i[i] + 1 \bmod (3) \rightarrow p_k \in all_seen_i$

Suppose that, during a delicate replacement, there exist a processor p_k that has a proposal with a phase value that is one step ahead of p_i's proposal. In that case, the identifier of p_k needs to be included in the set *allSeen*.

Invariant 2

$\exists c \in R : all_i[k] = true \wedge \exists c' \in R : all_i[k] = false \rightarrow phs_i[k] \in c \neq phs_i[k] \in c'$

Suppose that the identifier of p_k is included in the set *all*. In the case that one asynchronous round later, the same identifier is no longer included in *all*, then the phase value of p_k's proposal needs to have increased by one.

Invariant 3

$\exists c, c' \in R : phs_i[i] \in c \neq phs_i[i] \in c' \rightarrow all_i[i] \in c' = false$

Suppose that after one asynchronous round, the phase value of p_i's proposal has increased by one. In that case, the identifier of p_i can not be included in the set *all*.

Invariant 4

$\exists c, c' \in R : \forall p_k \in FD[i].part : echo_i(k) \wedge allSeen_i()$ holds in c $\rightarrow phs_i[i] \in c \neq phs_i[i] \in c'$

Suppose that the functions $echo()$ and $allSeen()$ both returns true for processor p_i. One asynchronous round later, the phase value of p_i's proposal needs to have increased by one.

Invariant 5

$phs_i[i] \neq 0 \rightarrow !allowReco_i()$

Given that the phase value of p_i's proposal is not equal to 0, the function $allowReco()$ should not return true.

Invariant 6

$\exists c' \in R : phs_i[k] = 2 \wedge all_i[k] = true \rightarrow \exists c \in R : FD_i[j].part \subset (allSeen_i \cup p_j)$

Given that during the current asynchronous round, the phase vlaue of p_i's proposal changed to 2 and that the identifier of p_i is as of this round no longer included in the set all, p_i's participant set is a subset of the union $allSeen$ and p_i's identifier.

Invariant 7

$\exists c \in R : \forall p_k \in P : phs_i[k] = 2 \rightarrow \forall p_k in P : prp_i[k] = prp_i[i] \in c$

Given that the proposal of every processor in the current configuration has a phase value that is 2, the set attached to every single proposal is the same.

Invariant 8

$\exists c, c' in R : \forall p_k \in FD[i].part : phs_i[k] \in c' \neq phs_i[k] \in c + 2 \bmod (2)$

It should not be possible for the phase value of a proposal to skip one step.

Invariant 9

$\exists c, c' in R : \forall p_k \in P : phs_i[k] \in c' \neq phs_i[k] \in c - 2 \bmod (2)$

It should not be possible for the phase value of a proposal to go back one step.

Invariant 10

Changes to degree by a call to $estab_i()$ should only be from 1 to 2

Invariant 11

Changes to degree by a call to $increment_i()$ should only be from 3 to 4, or from 5 to 0

Invariant 12

$\exists c, c' \in R : phs_i[i] \in c \neq phs_i[i] \in c' \rightarrow \forall p_k in FD[i].part : degree_i(i) \leq degree_i(k) \bmod 6$

For processor p_i to be able to change the phase value attached to its proposal in the current asynchronous round, every other processor should have a degree that is similar to p_i's, or a degree that is higher.

Invariant 13

$\exists c, c' \in R : config_i[i] \in c \neq config_i[i] \in c' \rightarrow degree_i(i) \in c = 5$

For a processor p_i to be able to change its own $config$-set, it needs to have a degree that is equal to 5.

Invariant 14

$\exists c \in R : \forall p_k \in FD[i].part : phs_i[k] = 0 \rightarrow \forall p_k \in FD[i].part : config_i[k] = config_i[i]$

Given that a delicate replacement just took place, and every processor in the previous configuration has changed its phase value from 2 to 0, every processor in the previous configuration should have the same $config$-set.

Invariant 15 (Corollary 4.14)

$\exists c \in R : (1, false) = (phs_i[i], myAll(i)) \in NA(c) \rightarrow$ either a call to $config_i(*empty_set*)$ is made $\vee\ \forall p_k \in FD[i].part : prp_i[k] = (0, *empty_set*)$

Suppose that the phase value of p_i's proposal is equal to 1 and and the identifier of p_i is not included in all. Within $\mathcal{O}(N)$, there should either be a call made to the $config()$-function, with the input set being the empty set, or the system should reach a state with no notifications.

Invariant 16

$\forall p_k \in FD[i].part : degree(k) \in \{0, 1, 2, 3, 4, 5\}$

Changes done to the degree are modulo six

Invariant 17

Changes made to $prp_i[i]$ should only be done via $estab_i()$, $maxNtf_i()$, $increment_i()$

Invariant 18

$\exists p_k \in FD[i].part : phs_i[k] = 0 \rightarrow prp_i[k] = (0, *empty_set*)$

If the phase value of p_i's proposal is equal to 0, then the set that is attached to p_i's proposal must be empty

Invariant 19

$N(c)$ should not contain a proposal other what have been proposed via a call to the estab()-function

Invariant 20

Changes to $config$ should only be done via line 22, line 24, line 29 or line 31

Invariant 21 (Theorem 4.16)

Within $O(n)$ rounds, no stale information of any type should be contained in the system

Invariant 22 (Theorem 4.9)

Let R be an admissible execution w.r.t participant of the Reconfiguration Assurance layer (which may include explicit delicate or spontaneous replacements). Within $\mathcal{O}(N)$ asynchronous rounds, R has a suffix that does not include a system state and $p_i, p_j \in P$ for which the following Equation 1 holds.
$(\exists p_k \in FD_i[i].part : \neg corrDeg(i, k)) \vee$
$(\exists p_k \in FD_i[i].part : (prp_j[k].phase = prp_j[j].phase + 1) \bmod 3 \wedge (p_k \not\ni allSeen_j))$

Invariant 23 (Lemma 4.7)

Stale information of type 4 will be removed within $\mathcal{O}(1)$ asynchronous rounds

Invariant 24 (Lemma 4.1)

Stale information of type 1 will be removed within $\mathcal{O}(1)$ asynchronous rounds

Invariant 25 (Lemma 4.2)

Stale information of type 2 will be removed within $\mathcal{O}(1)$ asynchronous rounds

Invariant 26 (Claim 4.6)

Suppose that in R's starting system state, it holds that for every two processors $p_i, p_j \in P$ that are active in R, we have that $(\{config_i[i], config_j[i], m_(i,j).config\} \backslash \{\perp, FD_i[i]\}) = \emptyset$, where $m_(i,j)$ is a message in the channel from p_i to p_j. Within $\mathcal{O}(1)$ asynchronous rounds, the system reaches a state in which $config_i[i] = FD_i[i]$.

Lemma 4.8

Let R be an execution of Algorithm 4.1 that is admissible with respect to the participant sets and that does not include an explicit (delicate) replacement. Suppose that in R's starting system state, c, there are notifications (of configuration replacement proposals)m i.e., $N(c) \neq \emptyset$. Moreover, suppose that for any active participant $p_i \in P$ and any $c* \in R$ it holds that $(prp_i[i], myAll_i(i)) = (n_i*, a_i* \in NA(c*)$. Within $\mathcal{O}(1)$ asynchronous rounds, the system reaches a state $c^{\circledast} \in R$ in which one of the following is true:
(i) $(n_i*, a_i*) \notin NA(c^{\circledast})$
(ii) the system takes a step (immediately after $c^{\circledast}$) in which there is a call to the function $configSet(\perp)$
(iii) the invariants (27) to (33) hold.

Invariant 27 (Lemma 4.8, part 1)

Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state $c' \in R$ in which for any $p_j \in FD_i.part$ that is an active participant in R, it holds that $(prp_j, all_j[j]) = (n_i*, a_i*)$ and $FD_j[i].part = FD_i.part$ and $FD_j[j].part = FD_i.part$. Moreover, $prp_i[i] \rightsquigarrow_{prp} prp_j[j]$ or $prp_j[j] \rightsquigarrow_{prp} prp_i[i]$*

Invariant 28 (Lemma 4.8, part 2)

Suppose that invariant (1) holds in every system state of R. Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c' \in R$*, in which it holds that* $echo_i[j].prp = n_i*$, $echo_i[j].part = FD_i[i].part$.

Invariant 29 (Lemma 4.8, part 3)

Suppose that invariants (1) and (2) hold in every system state of R. Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c' \in R$*, such that* $myAll_i(i) = true$ *holds in* c'.

Invariant 30 (Lemma 4.8, part 4)

Suppose that invariants (1), (2) and (3) hold in every system state of R. Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c' \in R$*, in which it holds that* $all_j[i] = true$*, where* $p_j \in FD_i[i].part$.

Invariant 31 (Lemma 4.8, part 5)

Suppose that invariants (1) to (4) hold in every system state of R. Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c' \in R$*, in which it holds that* $echo_i[j] = (FD_i[i].part, prp_i[i], myAll_i(i))$*, where* $p_j \in FD_i[i].part$.

Invariant 32 (Lemma 4.8, part 6)

Suppose that invariants (1) to (5) hold in every system state of R. Within $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c' \in R$*, in which it holds that* $p_i \in allSeen_j$*, where* $p_j \in FD_i[i].part$

Invariant 33 (Lemma 4.8, part 7)

Suppose that invariants (1) to (6) hold in every system state $c' \in R$*. Within* $\mathcal{O}(1)$ *asynchronous rounds, the system reaches a state* $c'' \in R$*, such that there exists an active participant* $p_k \in P$ *for which the if-statement condition of line 32 holds in* c''*. Specifically,* $\forall p_k \in FD_j[j].part : degree_j(k) - degree_j(j)(mod6) \in \{-1, 0, 1\}$ *holds in* c' *and either* ***(7.1)*** $\forall p_k \in FD_j[j].part : degree_j(k) - degree_j(j) \in \{0\}$ *and this part holds for any such* p_k*, or* ***(7.2)*** *the if-statement condition of line 32 holds in* c'' *for any* $p_k \in FD_j[j].part : degree_j(k) - degree_j(j)(mod6) \in \{-1\}$*, but not for* $p'_k \in FD_j[j].part : degree_j(k) - degree_j(j)(mod6) \in \{0, 1\}$

Invariant 34 (Lemma 4.12)

Suppose that there is a processor $p_j \in FD_i[i].part$ *that changes* $(prp_j[j], all_j[j])$ *more than thirteen times. In this case, within* $\mathcal{O}(N)$ *asynchronous rounds during* R'*, the system reaches to a state* $c^* \in R'$ *in which the following Equation holds.*

$$\forall p_k, p_j \in FD_j[j].part : ((prp_k[\ell] = prp_j[j]) \wedge ((prp_j[k], all_j[k]) = (prp_j[j], all_j[j] = true)))\wedge$$
$$(\exists m \in channel_{j,k} \cup channel_{k,j} : j \in \{k, \ell\} \longrightarrow m = (\bullet, prp_j[j], all_j[j] =$$
$$true, (\bullet, prp_j[j], all_j[j] = true)))\wedge$$
$$(\exists m \in channel_{k,\ell} : j \not\ni \{k, \ell\} \longrightarrow m = (\bullet, prp_j[j], \bullet, (\bullet, prp_j[j], \bullet)))\wedge$$
$$(p_j \in allSeen_k) \wedge (all_j[j] \wedge FD_j[j].part \subset (allSeen_j \cup \{p_j\}))$$

Invariant 35 (Theorem 4.17)

Let R be an execution of Algorithm 4.1. Suppose that execution R starts from a system state, c, that includes no stale information. (1) For any system state $c \in R$*, it holds that c includes no stale information. Suppose that the step that immediately follows c includes a call to estab(). (2.a) The only way that set becomes a notification is via a call to estab(set) (line 7). (2.b) The only way that a processor becomes a participant is via a call to participate() (line 8). (3) If notifications exist, then the configuration is replaced within* $\mathcal{O}(N)$ *asynchronous rounds. (4) The requirements that appear in Figure 1 hold during R.*
The above is taken from [1], thus all references are for figures and rows are to that specific paper.

Invariant 36

Arriving messages to active processors should only be sent from processors considered to be participants

Invariant 37

Upon a delicate replacement, all active participants will eventually have the same proposed set