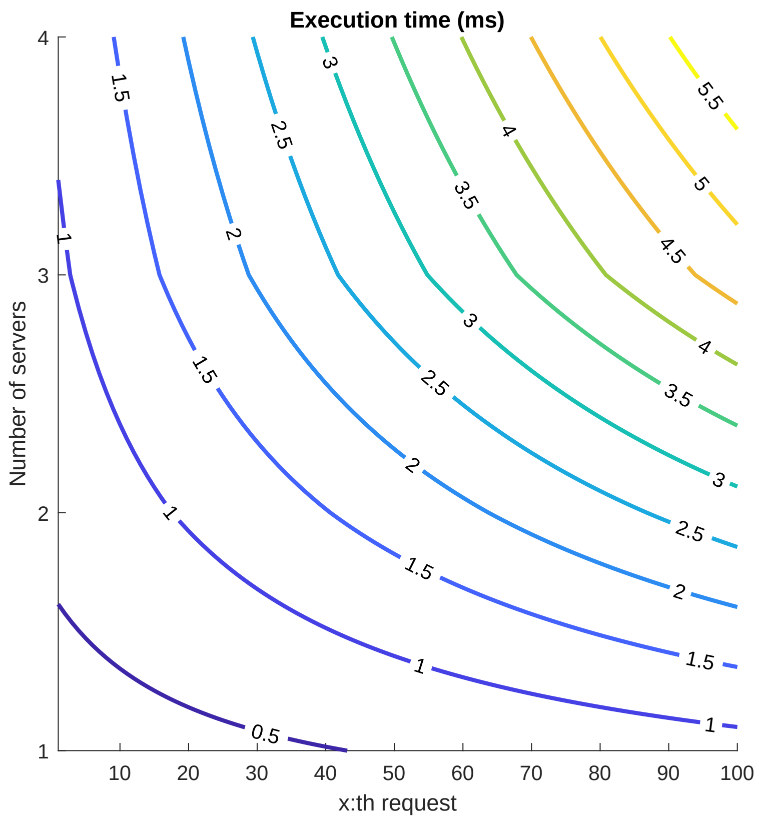

Self-Stabilizing Byzantine Fault-Tolerant State Machine Replication

Rust Implementation, Experimental Evaluation and Applications in Trucks

Master's thesis in Computer science and engineering

Chibin Kou
Oskar Lundström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Self-Stabilizing Byzantine Fault-Tolerant State Machine Replication

Rust Implementation, Experimental Evaluation and Applications in Trucks

Chibin Kou

Oskar Lundström

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Self-Stabilizing Byzantine Fault-Tolerant State Machine Replication
Rust Implementation, Experimental Evaluation and Applications in Trucks
Chibin Kou
Oskar Lundström

Supervisor: Elad Michael Schiller, Department of Computer Science and Engineering
Advisor: Styrbjörn Törngren, Volvo Technology AB
Examiner: Tomas Olovsson, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Execution time of the x:th request for various number of servers, on an embedded evaluation environment consisting of Raspberry Pis. It is the same as Figure 4.5.

Typeset in LaTeX
Gothenburg, Sweden 2020

Self-Stabilizing Byzantine Fault-Tolerant State Machine Replication
Rust Implementation, Experimental Evaluation and Applications in Trucks
Chibin Kou
Oskar Lundström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern society relies on many critical digital services such as cloud storage and vehicular systems. To bring reliability to these services, State Machine Replication (SMR) is one possibility. This thesis focuses on the performance evaluation of the SMR algorithm by Dolev *et al.*. Specifically, our evaluation criterion considers latency, which is the time from when a client issues a request until it has received replies from the servers. We base our study on an analytical estimation of the latency as well as a high-quality pilot implementation in the Rust programming language, which is suitable for embedded systems. Moreover, since Dolev *et al.* consider both synchronous and asynchronous system executions, we study our pilot implementation on PlanetLab with 15 nodes and 26 ms message round trip delay as well as an embedded system with 5 nodes and 0.17 ms message round trip delay. We find that the SMR algorithm by Dolev *et al.* scales linearly with the number of completed requests, the number of servers as well as the number of clients. Furthermore, we also find that the performance behavior is similar in the two environments, although the network latency plays a big role. In addition to the above, we have also identified three possible use cases of SMR in trucks.

Regarding the use of unbounded variables, queues, and message sizes that appear in the SMR algorithm by Dolev *et al.*, we offer the use of a global reset algorithm by Georgiou *et al.* In addition to a detailed evaluation of the algorithm, we explain how this self-stabilizing algorithm can fit in the analytical framework of Dolev *et al.*, which considers both Byzantine faults and arbitrary transient faults. Our evaluation shows that the latency of the global reset algorithm is around four times the slowest network round trip time of the participating nodes.

Keywords: Automotive, Byzantine Fault-Tolerance, Distributed Systems, Self-Stabilization, State Machine Replication.

Acknowledgements

We want to thank our Chalmers supervisor Elad Michael Schiller, who has worked closely with us and has through his expertise provided us with tremendous help in this project.

We also want to thank our Volvo advisor Styrbjörn Törngren for helping us throughout this project, and especially for sharing his insights on the industrial aspects of the thesis.

Lastly, we want to thank our Volvo manager Leif Frendin for welcoming us to his team and providing us a place to work at Volvo.

Chibin Kou & Oskar Lundström, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1

Introduction

Modern society relies on many critical digital services such as cloud storage and vehicular systems. For these critical services, fault tolerance is a highly desirable property in order to provide continuous availability, safety and correctness. To this end, some form of redundancy can be added to the system so that if one component fails, there are other components ready to take over. One way to achieve fault tolerance and redundancy is through the use of distributed systems. In a distributed system, there are several nodes that collaborate in order to accomplish a common task, and they communicate by sending messages to each other.

There are several types of faults that can occur in digital services. To this end, algorithms can have various fault tolerance guarantees. One type is *Byzantine fault tolerance* [60]. A Byzantine fault is when a node starts behaving arbitrarily, possibly actively trying to prevent the system as a whole to work correctly, by not following the system's specification. Byzantine faults can happen due to for example software bugs or malicious attacks. Another fault tolerance property is *self-stabilization* [21]. A self-stabilizing system can start from any arbitrary initial state, and still automatically recover to a legal state, where the system performs the task it is supposed to do. Thus a self-stabilizing system can recover from arbitrary transient faults, such as memory corruption due to electromagnetic interference. It also has the potential of saving costs, because the system recovers automatically, without any administrator having to take any action, which saves manpower.

In this Master's thesis project, we study a general and powerful tool called *State Machine Replication* (SMR), which has been extensively researched in the distributed systems literature. State machine replication is a general technique that can be used for many programs, including non-trivial real-world distributed systems like NFS (Network File System), ZooKeeper [57] and Bitcoin [66]. It is worth noting that ZooKeeper is not Byzantine fault tolerant while Bitcoin is. In particular, we focus on a state machine replication algorithm with especially high fault tolerance guarantees, namely, the state machine replication algorithm by Dolev *et al.* [22], which is Byzantine fault tolerant and self-stabilizing.

This thesis focuses on the performance evaluation of [22]. Specifically, our evaluation criterion considers latency, which is the time from when a client issues a request until it has received replies from the servers. We base our study on an analytical estimation of the latency as well as a high-quality pilot implementation in the Rust programming language, which is suitable for embedded systems. Moreover, since [22] considers both synchronous and asynchronous system executions, we study our pilot implementation in PlanetLab with 15 nodes and 26 ms message round trip delay as well as an embedded system with 5 nodes and 0.17 ms message round trip delay. Regarding the use of unbounded variables, queues, and message sizes that appear in [22], we offer the use of a global reset algorithm by Georgiou *et al* [51]. In addition to a detailed evaluation of the algorithm, we explain how this self-stabilizing algorithm can fit in the analytical framework of [22], which, as already mentioned, considers both Byzantine faults and arbitrary transient faults.

In the following sections, we present a brief introduction to state machines and state machine replication as well as the purpose, related work and contributions of this project.

1.1 State machines and distributed replication

A state machine is a very powerful mathematical abstraction of a computation in computer science. It consists of *states* and *transitions*. States, as the name suggests, represent the observable internal status of some computation. Transitions are triggered by external inputs to the state machine and cause it to transit from one state to a new state, which might or might not be the same as the old state. One intuitive example that helps explain this concept is the use of data structures in programming languages, which are essentially state machines. The data stored inside the data structure represents the state and the call to the data structure's API constitutes transitions that can modify the state.

The concept of state machines is vital in computer science because it is the foundation of many advanced technologies, such as regular expressions and parsers of compilers. In a broader sense, it helps simplify the problem that computer scientists are trying to solve. For example, if we want to solve some difficult problem which can be modeled as a state machine, and we have a general solution for state machines, then the problem is automatically solved. In the context of this thesis, the problem in question is how to add redundancy to a service to increase reliability. If we can model the service as a state machine, and we have a solution for adding redundancy to a state machine, which is exactly what SMR does, then the problem is solved.

Now the question is how to add redundancy to a state machine. The answer to this question is presented in the rest of the thesis. But first we need to present the underlying concept being used, which is *distributed replication*. The concept can be simply explained as having multiple identical backups of something of great importance in a distributed manner. In the case of SMR, we are replicating the same state machine in distributed servers. In the next section we further explain the idea and some challenges it imposes.

1.2 A brief introduction to state machine replication

SMR is a general method for adding redundancy to programs that are deterministic and follow the client-server model. These programs can range from simple ones such as a dictionary with the operations `insert`, `get` and `remove`, to more complex ones such as NFS. The power of SMR is that the program to be replicated does not have to consider redundancy by itself at all. In order to understand SMR, let us first consider a system without SMR, depicted in Figure 1.1.

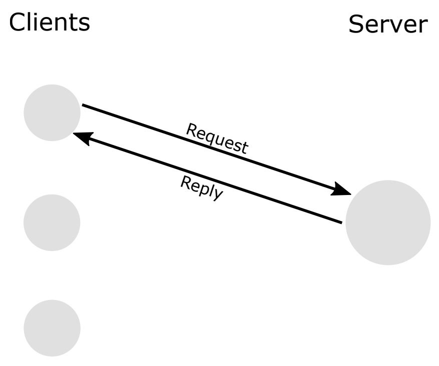

Figure 1.1: A diagram of a system without SMR.

In Figure 1.1 we see a single server that runs the server-side of the program. There are multiple clients that can send requests to the server. However, this single server constitutes a single point of failure, since if this server crashes, the service provided by the program is no longer available to the clients.

In order to add redundancy, we need to add more servers. A naive approach would be to simply add more servers that all run the program in question, let the clients send their requests to all servers and then take any of their replies. However, there is a serious flaw with this approach, namely, it can happen that different servers receive the requests in different orders. Furthermore, some servers might not receive all requests. This inconsistency means that the servers are in different states, which is undesirable. For example, it could mean that a file only exists in some of the servers but not the others.

The solution to the above inconsistency problem is SMR. Without going into all details, SMR essentially acts as a middle layer between the clients and the servers, making sure that all serves actually do receive all requests and in the same order. How this happens is up to the specific SMR algorithm in question, and for all details, we refer to [22], which describes how the SMR algorithm this thesis focuses on works.

At this point, we have multiple servers that the SMR algorithm makes sure are in identical states. If one or more of them crashes, other servers are alive and the service is still available to the clients. In fact, the SMR algorithm is completely transparent to the clients, meaning the clients do not notice any difference between accessing a single server directly, or multiple servers through SMR. In particular, this means that for the client, it does not matter that some of the servers have crashed.

To summarize, SMR is used to add redundancy to a program. This is achieved by running the program on multiple servers, and letting them receive client requests in the same order. SMR acts as a middle layer between the program's clients and servers and is completely transparent.

1.3 Purpose

The main purpose of this master thesis project is to evaluate the performance of the self-stabilizing Byzantine fault tolerant state machine replication algorithm of [22] (briefly described in Chapter 2). In order to do this, we provide an efficient real-world implementation of it (Chapter 3). Furthermore, we also conduct an experimental evaluation of it (Chapter 4) where we aim at answering the following research questions.

- How does the performance change as more client requests have been completed?
- How well does the system scale with the number of servers and clients?
- How does the system perform in different environments?

The knowledge we gain from the above is important, because it is useful to know how an algorithm performs in practice.

As a second purpose, we aim at finding possible applications of state machine replication in trucks (Chapter 5).

1.4 Related work

In the context of fault tolerant state machine replication, one of the most well-known algorithms is *Practical Byzantine Fault Tolerance* (PBFT) [15]. PBFT was the first state machine replication algorithm to be efficient, Byzantine fault tolerant *and* work in asynchrony, whereas earlier algorithms were either too slow to be used in practice, or they assumed synchrony. Following in the footsteps of PBFT, many variations of Byzantine fault tolerant state machine replication have been proposed, including BFT2F tolerating more faulty nodes [61], FaB reducing the number of communication rounds [4] and Zyzzyva improving performance with speculation

[59].

Another algorithm following the footsteps of PBFT is *Self-Stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors* [22], which is the algorithm this thesis focuses on. As the name suggests, [22] is a self-stabilizing variation of PBFT, which means that it can handle arbitrary transient faults without human intervention [21], something PBFT cannot do. [22] is the only self-stabilizing Byzantine fault tolerant state machine replication algorithm with the exception of [8]. The difference is that [22] is based on failure detectors while [8] is based on clock synchronization, and thus [22] requires weaker synchronization guarantees. We note that the class of algorithms designed for asynchronous systems is considered stronger than the ones designed for synchronous systems, because in the latter class all nodes need to wait for the slowest node in the system. We note that the implementation of [8] requires the implementation of a secure self-stabilization clock synchronization, such as [48, 55, 56].

Of special interest to this practically oriented project, it is worth noting that in addition to theoretical work, the literature already contains several implementations of state machine replication algorithms in practice. The already mentioned PBFT also included a real-world implementation and evaluation, where an NFS service was replicated using the PBFT algorithm. It was found that the overhead of state machine replication in that case was only 3%, meaning state machine replication is indeed feasible to use in practice. Another well-known real-world implementation is BFT-SMaRt [7]. BFT-SMaRt is a Java implementation of a PBFT-like protocol focusing on ease-of-use, reliability and making state machine replication available “to the masses” [7]. Finally, [22], the focus point of this project, has been implemented and practically verified in [67]. The implementation was done in Python with the aim of developing a proof of concept, used to validate the correctness of the algorithm in favor of focusing on performance. We mention that our performance-focused implementation is around two magnitudes faster than their implementation.

We note the existence of alternative algorithms for implementing state-machine replication using self-stabilizing group communications [23, 24, 25, 40, 41, 42] as well as self-stabilizing virtual nodes [9, 26, 27, 28, 29] and self-stabilizing consensus [32, 33].

One may see Byzantine fault tolerance as a way to model malicious behavior, which requires a bound on the number of faulty (Byzantine) nodes. In the context of self-stabilization, another kind of malicious behavior is considered, which model faulty nodes as rational ones [36, 37, 43, 44, 45, 46, 47]. There, however, no bound on the number of nodes is required.

In the context of self-stabilization and Byzantine-fault tolerance, we note the existence end-to-end communication [34, 35], which extends the fault model of the state-of-the-art self-stabilizing end-to-end communication algorithm [30, 31].

1.5 Contributions

We provide the first performance-focused implementation (Chapter 3) of the self-stabilizing Byzantine fault tolerant state machine replication algorithm by Dolev *et al.* [22] as well as the corresponding performance evaluation of it (Chapter 4). We find that the client latency of [22] scales linearly with the number of completed client requests, the number of servers and the number of clients. These results were achieved both on PlanetLab and an embedded environment consisting of Raspberry Pis.

Our second contribution concerns an analytical lower bound on the client latency of [22]. The bound, for $n \geq 3$ servers, is that a client request takes at least 5 message trips over the network to complete. We also confirm this bound in practice by comparing it to our experimental findings.

We offer an analytical latency estimation of the global reset algorithm [51] and compare it to our implementation and experimental evaluation of the algorithm. We find, analytically and experimentally, that the global reset time is 4 times the slowest network round trip time of all nodes in the system.

Finally, from the industrial point of view, we have identified three possible applications of state machine replication in trucks, namely ISO 26262 Diverse Redundancy, fault tolerant logging and distributed services (Chapter 5).

1.6 Organization

This report is organized as follows. Chapter 2 contains some theoretical background to state machine replication and related concepts that can facilitate a more complete understanding of the remainder of the report. Chapter 3 describes what our implementation of the SMR algorithm looks like. Furthermore, it also contains the design decisions we have made in order to achieve high performance. In Chapter 4 we present how we conducted our performance evaluation of the algorithm and our implementation of it, as well as the results. Chapter 5 contains three possible applications of state machine replication in trucks. Chapter 6 presents the Global Reset technique and our performance evaluation of it. Finally, we conclude the report in Chapter 7.

2

Theory

In this chapter we present the theory needed to understand the rest of the report: terminology and notation, state machine replication, fault models as well as the state machine replication algorithm we focus on.

2.1 Terminology and notation

Throughout the report, we use the following terminology and notation.

- **SMR:** an abbreviation of state machine replication.
- **Node:** an entity (e.g. a computer) that runs an instance of the SMR algorithm.
- **Server:** a node that runs the server-side of the algorithm.
- **Client:** a node that runs the client-side of the algorithm.
- n**:** the number of servers.
- f**:** the number of faulty servers.
- k**:** the number of clients.

2.2 State Machine Replication

A *state machine* is an abstract concept of computation which is modeled by a set of states and transitions. Given a state and an input, a state machine transitions to a new state, and so on. There are different types of state machines depending on the assumption. In this thesis, we focus on deterministic infinite state machines. Deterministic means that given the same state and input, applying the same transition always leads to the same resulting state. Infinite means that the states are not from a predetermined fixed set of states. This is a general type of state machines that can model a number of different applications, from simple data structures (such as a list) to entire programs (such as NFS [15]).

As the name suggests, *State Machine Replication* (SMR) is the action of replicating a state machine in several logical or physical locations, each of which is called a *replica*. These replicas should contain the same state, which is determined by the starting state and the transitions that have been applied. Therefore, in order to achieve consistency, all replicas need to be initialized with the same starting state, and the SMR algorithm should provide a mechanism which guarantees that all transitions are carried out in the same order at all replicas.

State Machine Replication is a general technique used for implementing fault tolerant services. To be more specific, SMR can be used as a way to add redundancy to the system. When a service, which is modeled as a state machine, is replicated in different locations, even if some of the replicas fail, there are still some surviving replicas that can continue to provide service. Moreover, SMR can also prevent so called Byzantine faults [60] (see Section 2.3) if implemented in a certain way. For example, if one replica is compromised by an attacker and returns incorrect replies to the client, the client can still obtain the correct state by comparing the replies it gets from other replicas. The assumption here is that an attacker cannot compromise a majority of all replicas.

From the users' perspective, services that use SMR under the hood is very similar to the client-server model [72, Chapter 2.3], as shown in Figure 2.1. The client first broadcasts a request to all replicas. Then the replicas communicate with each other in order to reach consensus about what the next state should be. Finally, the reply is returned to the client. The client can decide what to do with these replies based on the requirements of the application. For example, if the application does not need to provide strong consistency guarantees, i.e., it is not necessary for the user to get the newest state, then the client can just return the first reply it gets to the user. Therefore the user can get the result very fast but with the possibility of getting outdated state. However, if the application needs to fully exploit the strong fault tolerant guarantees that SMR can provide, then the client needs to collect enough replies to mask out the faults. It is worth noting that this procedure can be implemented in a client wrapper that handles everything for the user. In that case, the user does not notice any difference from using a service that is provided by a single server. We can also observe from Figure 2.1 that even if one of the replicas fail, the client can still get replies from the remaining replicas.

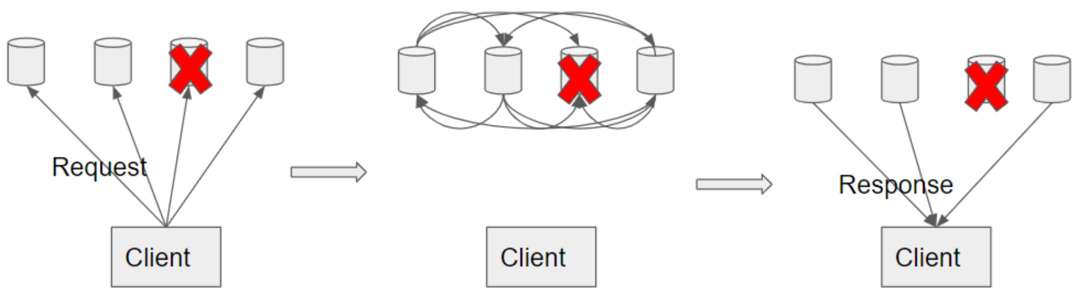

Figure 2.1: SMR service paradigm, the replica with a cross means it crashed

2.3 Fault models

One of the main challenges of SMR is how the system can provide services correctly in the presence of failures. The SMR algorithm that we focus on in this thesis can tolerate Byzantine and arbitrary transient faults. These two types of faults are described in this section.

Byzantine faults. Byzantine faults [60] represent a type of fault in distributed systems where some nodes fail and behave arbitrarily. In contrast to a crash failure, a Byzantine failure does not necessarily mean that the node stops sending messages. Instead, a Byzantine node can still be active and send arbitrary messages that do not follow the specification of the algorithm running on the system. The messages sent can simply be corrupted, or they can be carefully constructed by a malicious attacker whose aim is to cause to system to behave incorrectly.

Compared to crash failures, Byzantine failures are more difficult to handle since it is no longer possible to trust that the messages received follow the specification. In fact, it has been proven that in order to solve consensus [18, Chapter 15.5], the number of Byzantine nodes in the system must be less than one third [60]. Since many distributed algorithms are related to consensus, this one third ratio is common to find in many Byzantine tolerant algorithms.

Byzantine faults can occur due to several reasons [15]. As previously mentioned, they can be the result of a malicious attack, where an intelligent attacker compromises one or more nodes in the system and has control over what messages are being sent. However, Byzantine faults do not necessarily happen due to malicious reasons. They can also be the result of hardware and software bugs, where e.g. a buffer overflow in one node causes it to send corrupted or outdated messages.

Arbitrary transient faults and self-stabilization. An arbitrary transient fault represents any possible temporary fault that can happen to the system, except that the algorithm code stays intact [20, 21]. These types of faults include e.g. memory corruption induced by electromagnetic interference and control logic failures at the hardware level. The combination of all things that can go wrong puts the system in an arbitrary state, from which a *self-stabilizing* algorithm can recover. This is depicted in Figure 2.2.

Comparison. Byzantine faults and arbitrary transient faults can sound similar, but they are indeed different types of faults. Table 2.1 shows their two key differences. Byzantine faults are *permanent*, meaning that a Byzantine node acts in an arbitrary way during the entire lifetime of the system. Arbitrary transient faults on the other hand are *temporary*, meaning that they only appear once, and then the nodes start behaving according to the algorithm again. However, once the nodes start behaving according to the algorithm, they start from an arbitrary state, which the algorithm must handle. Since Byzantine faults are permanent, there must be a limit on how many nodes that are allowed to be Byzantine and yet still allow the

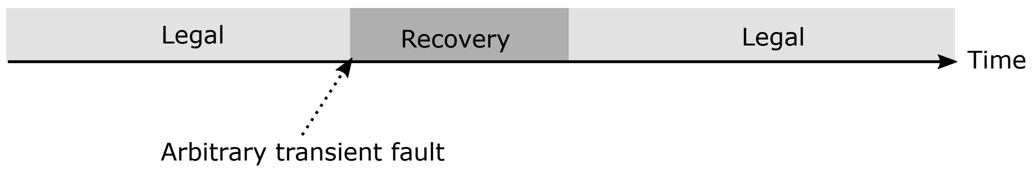

Figure 2.2: A time line of an execution of a self-stabilizing algorithm. First, the execution is legal (i.e. the system does what it is supposed to do). Then an arbitrary transient fault occurs. Immediately after this event, the system enters a period where it recovers from the fault. After this recovery period, the system execution is legal again.

system to behave correctly. As previously mentioned, this limit is that less than one third of all nodes may be Byzantine. In contrast to Byzantine faults, arbitrary transient faults may affect all nodes in the system and the system can still function correctly after that, provided it is self-stabilizing.

Table 2.1: A comparison between Byzantine faults and arbitrary transient faults. Byzantine faults are permanent but only less than one third of the nodes may have this fault. Arbitrary transient faults are temporary but all nodes may experience this fault.

	Duration	**Nodes with the fault**
Byzantine fault	Permanent	Less than one third
Arbitrary transient fault	Temporary	All

2.4 Self-Stabilizing Byzantine Fault Tolerant Replicated State Machine Based on Failure Detectors

In this section we give a broad overview of the SMR algorithm by Dolev et al. [22], which is the algorithm this thesis has implemented and evaluated. This algorithm is both Byzantine fault tolerant and self-stabilizing. We focus only on giving an overview, and some more details on the parts that are critical for the rest of the report. For all details, we refer to the paper itself [22].

The algorithm consists of three main modules, the *View Establishment* module, the *Replication* module and the *Primary Monitor* module. Each of these modules has a specific responsibility, and provides information to the other modules. Figure 2.3 shows a simplified structure of the algorithm and the relationships between the modules.

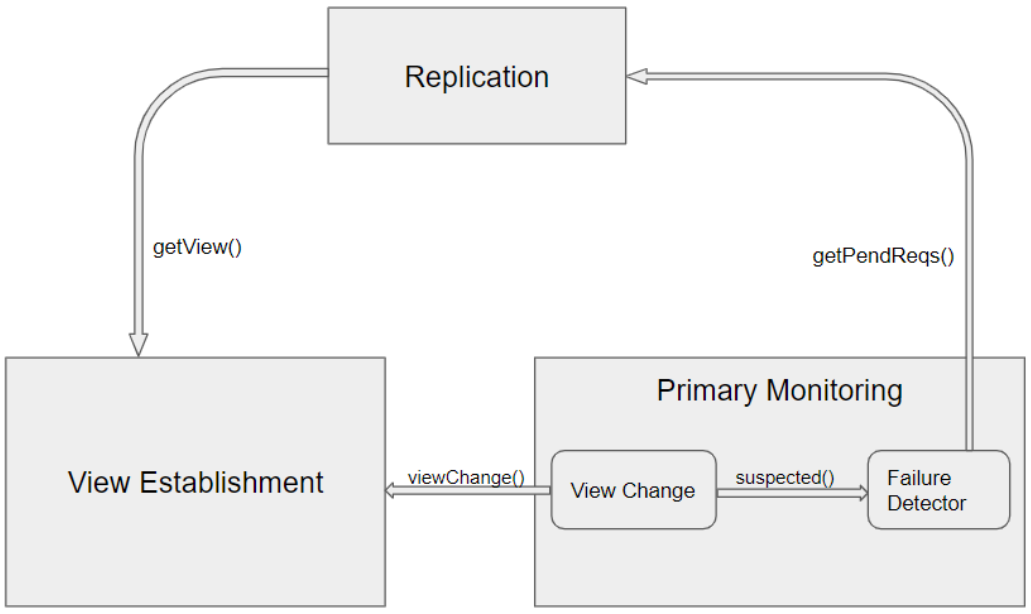

Figure 2.3: A simplified structure of the algorithm drawing inspiration from [22, Figure 1]
. An arrow from A to B indicates that module A calls a function on module B.

View Establishment. View Establishment is responsible for establishing a unique and shared *view* among all correct nodes. A view defines which the current *primary* node is, which is responsible for sequencing the client requests. Replication retrieves this view with `getView()`. If Primary Monitor suspects that the primary is not functioning as expected, a *view change* is carried out by View Establishment, letting all correct nodes agree on a new view, i.e., a new primary that is functioning. This happens when `viewChange()` is called on View Establishment.

Replication. Replication is responsible for the actual state machine replication task. To do this, the primary assigns a unique sequence number to each client request. Then, all requests are executed in the same order at all nodes, which is defined by these sequence numbers. Simply put, this module guarantees that all correct nodes have the same state in their state machine. In order to understand the evaluation results of Chapter 4, it is important to know that the local state of Replication contains an unbounded list holding *all* client requests that have ever been executed in the system. This list plays a big role on the performance.

Primary Monitor. Primary Monitor is responsible for monitoring the behavior of the primary and replacing it if needed. It consists of two sub-modules, View Change and Failure Detector. If Failure Detector suspects that the primary is misbehaving, it notifies View Change of this with a call to `suspected()`, upon which View Change raises a "need change" flag. If View Change sees that sufficiently many nodes have raised this flag, View Change instructs View Establishment to perform the previously described view change.

The second sub-module of Primary Monitor, Failure Detector, performs two checks: a *responsiveness check* and a *progress check*. The responsiveness check is carried out using a failure detector [49, 69], which works as follows. Node p_i maintains an integer array for all servers in the system. Whenever node p_i receives a message from p_j, node p_j's entry in the array is set to zero, and all other entries are incremented by one. Since a crashed node never sends messages, entries belonging to crashed nodes are never set to zero and eventually exceed a threshold, and in this way, p_i can distinguish crashed nodes from alive nodes. The progress check is carried out by checking that two consecutive calls to `getPendReqs()` return different results. If so, the primary has made progress. This check is to prevent the case where a Byzantine primary sends dummy messages but is not progressing the replication.

Module structure and communication pattern. All three modules have the same basic structure and communication pattern. They are all based on a *do forever loop* that runs the algorithmic logic forever. In the end of this loop, they send their entire local state to all other servers in the system. In particular, this means that the unbounded list of Replication previously described is sent in every Replication message. Furthermore, Primary Monitor uses a *token*, which carries some information used in order to allow Primary Monitor to detect failures. This token is piggybacked on every message sent by the algorithm.

3

Implementation

In this chapter we describe the design details of our pilot implementation. We describe some programming language aspects, the algorithmic layer, threads and communication, utility programs as well as new and reused code.

3.1 The Rust Programming Language

For our implementation, we have used *The Rust Programming Language* [58, 65]. Rust was chosen because of its strong type system, memory safety and high performance. Since the data types in [22] are non-trivial, Rust's strong type system facilitated making the data types clear and avoiding type issues. Because [22] aims to be very fault tolerant, we found it appropriate to use a language with strong memory safety in order to further facilitate fault tolerance. Unlike C/C++, Rust avoids issues such as dereferencing null, dangling pointers and uninitialized variables at compile time, which prevents human programmers from making those mistakes. Finally, like C/C++, Rust is a compiled language, which contributes to its high performance. Furthermore, unlike Java, Rust does not have a runtime garbage collector that can negatively impact the performance.

A *channel* [58] is a synchronization primitive existing in several programming languages, including Rust, for writing multi-threaded programs. A channel is used for sending data from one thread to another in a safe manner and allows two threads to coordinate their actions without using shared memory and locks. The channel primitive makes the producer-consumer pattern [6] easier to implement, which made channels a good fit for our implementation of [22], since as Section 3.3 shows, our implementation is based heavily on the producer-consumer pattern.

In this implementation, we have mainly used a particular flavor of channels called *ring channels*[1]. A ring channel is a channel that is FIFO-ordered, has bounded capacity and is non-blocking by overwriting. As Section 3.3 shows, these are the properties needed in our implementation.

Central to Rust is *Cargo*. Cargo is Rust's build system, package manager and testing

[1]`https://crates.io/crates/ring-channel`

utility, and thus handles all basic needs for programming in Rust.

3.2 The algorithmic layer

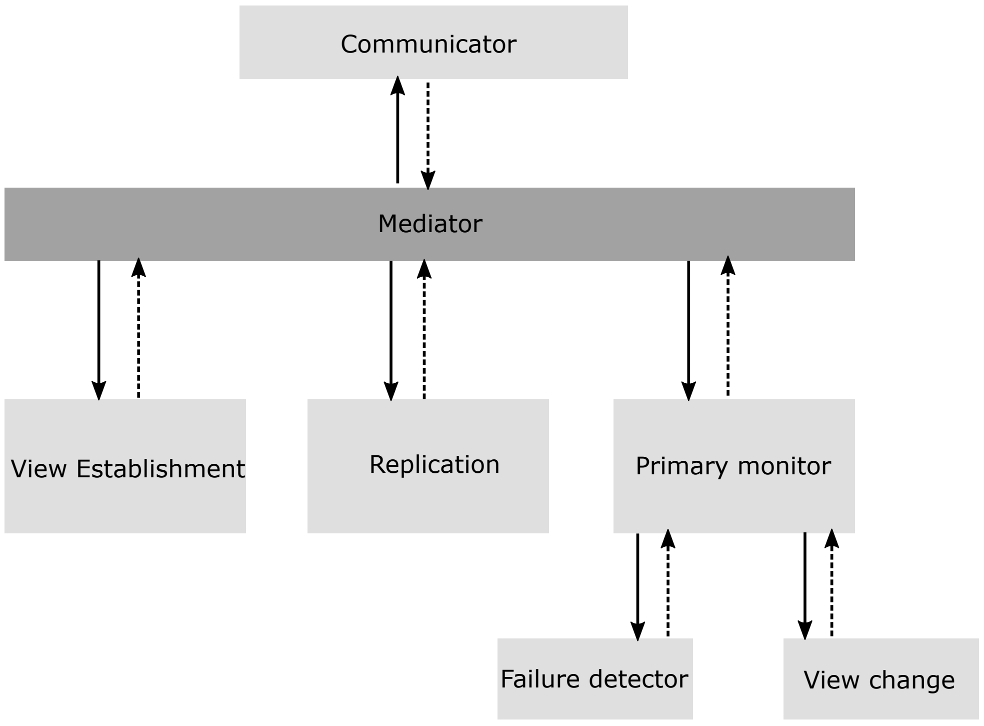

Figure 3.1: An overview of the implementation design. A solid arrow from A to B represents that struct A owns struct B. A dashed arrow from A to B represents that struct A has a reference to struct B. Mediator is the central struct and is the starting point of the program.

Figure 3.1 shows an overview of the algorithmic layer in our implementation, depicting the implementation's modules and their interaction. It is useful to have Figure 2.3 of Section 2.4 in mind when looking at this figure. The main module is *Mediator*, which is also the entry point of the program. Mediator owns the modules *View Establishment*, *Replication*, *Primary Monitor* and *Communicator*. The first three correspond to the three modules of the algorithm in [22] and are described in Section 2.4. View Establishment as described by [22] consists of the two modules *Coordinating Automaton* and *Predicates and Actions*. However, since the interaction between these two modules is very evolved, we implemented them as only a single module. For Primary Monitor, on the other hand, we followed [22] and implemented its two modules *Failure Detector* and *View Change* as separate modules. Communicator does not appear explicitly in the paper, but is needed in the implementation to handle communication.

As evident from Figure 2.3 and [22, Figure 1], all three modules of [22] interact heavily with each other, with function calls in both directions between each of them. One way to implement this would be to have direct references between the different modules. Due to the circular references between the modules, it is not possible to have (strongly typed) type-generic references *and* static dispatch in Rust. It is desirable to have type-generic references inside the modules to increase modularity and facilitate testing. Moreover, static dispatch is preferable to dynamic dispatch from a performance point of view. This meant that direct references between the modules were ruled out and is the reason why the *Mediator Design Pattern* [50] was chosen.

In the mediator design we used, the three modules have references to Mediator instead of each other. Furthermore, all inter-module interaction happens through Mediator. For example, when Replication wants to call `getView()` on View Establishment, Replication calls `getView()` on Mediator, which in turn calls `getView()` on View Establishment and returns the result to Replication. A similar technique is used for Primary Monitor's Failure Detector and View Change modules. All modules communicate using Communicator through Mediator as well.

3.3 Threads and communication

The program is based on threads and handles communication of the program is based on UDP. UDP was chosen instead of TCP for its simple use and performance. With UDP, each application instance needs only one socket, compared to one socket per node, if TCP was used. The algorithm in [22] is based on sending messages between the nodes. This means that datagram sockets (UDP) fit better than stream sockets (TCP). Furthermore, UDP offers better performance than TCP, especially in this case, since due to the way the algorithm in [22] is designed, re-transmissions are not needed in most cases. This is described more in Section 3.3.1. Our thread-based implementation processes incoming and outgoing messages in a way that emulates asynchronous I/O, as will become apparent in this section.

Figure 3.2 shows an overview of the threads used and how they interact. The program uses three threads: the *Sender* thread, the *Processing* thread and the *Receiver* thread. Sender serializes messages into bytes and sends them on the single UDP socket of the program. Receiver receives messages from the same UDP socket and deserializes them from bytes. Processing runs the actual algorithmic layer. In other words, of the modules of Figure 3.1, Processing runs everything except Communicator. Instead, Sender runs the send-side, and Receiver runs the receive-side, of Communicator. The inner workings of the three threads are described in more detail in sections 3.3.2, 3.3.3 and 3.3.4.

The threads interact mainly with channels. As Figure 3.2 shows, the threads are connected by multiple channels: one per message type and node (the exception being between Sender and Receiver for Replication and Primary Monitor messages). Processing sends a message by putting it on the send channel corresponding to the

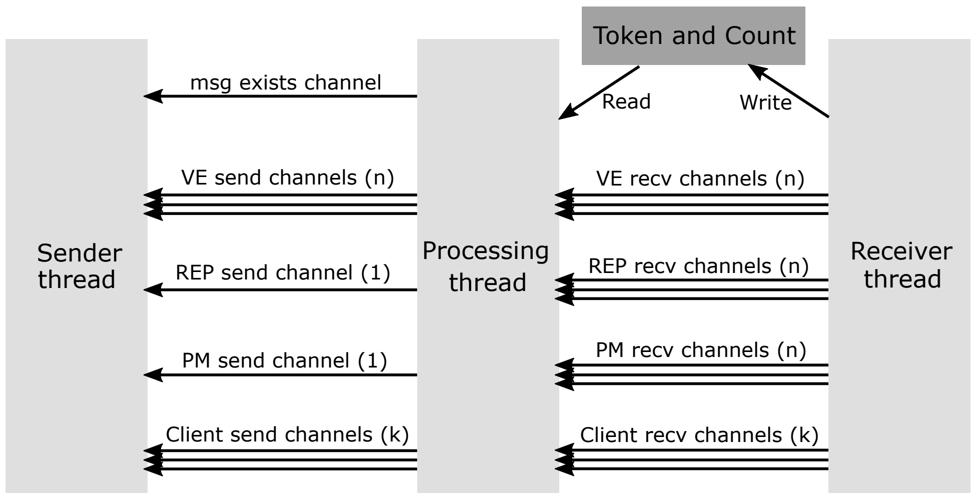

Figure 3.2: An overview of all threads and the coordination between them. The horizontal arrows represent channels. The numbers in parenthesis are the number of channels in that group.

message type and receiver node. For Replication and Primary Monitor messages, the channels are not separated by nodes, because identical messages shall be sent to all nodes for those message types. After a message is put on the channel, Sender handles the message at its own discretion. Processing receives a message, at a time it decides, by taking a message from the receive channel corresponding to the message type and node it wants to receive from. We note that the time by which a message is received by Receiver, and the time by which it is received by Processing, can be different. Similarly to Sender, Receiver handles incoming messages at its own discretion and simply makes sure to supply Processing with messages.

The channels described above carrying server messages have capacity 2 messages and the channels carrying client messages have capacity 100 messages. For the channels carrying server messages (View Establishment, Replication and Primary Monitor), the capacity is 2 since upon reception of a server message, the algorithm in [22] completely overwrites the old values. Due to this, only the most recent version of a message is needed. Capacity 2 was chosen instead of 1, because with capacity 1 we noticed a significant loss of performance, which might be due to race conditions on the single element of the channel. The channels carrying client messages have a larger capacity of 100 messages. The reason is that a more recent client message should not overwrite an older client message. With capacity 100, we experienced that Processing has time to handle client messages before they are overwritten.

The channels are separated by node and message type for fairness reasons. The channels have to be bounded, so that a Byzantine node cannot make them arbitrarily long and delay the processing of legitimate messages for an arbitrary amount of time. But if the channels are bounded and not node-separated, a Byzantine node can instead cause legitimate messages to be overwritten by sending messages at a high rate. By separating the channels per node, this problem is avoided. The channels are also separated by message type so that the three modules can be run at different speeds. Even if one module is run much slower than the other two, its messages are not overwritten by other messages types since they are on separate channels. In our current implementation, the three modules run at the same speed, but with this separation, future optimizations are easier.

There are several advantages of the design of Figure 3.2. One advantage is that there is no need for concurrency control (locks) inside Processing. This simplifies development, debugging and testing. Furthermore, the lack of locks increases performance. By moving message serialization/deserialization out of Processing, Processing is offloaded of those tasks. When serialization/deserialization happen on two separate threads, more cores of the CPU are utilized, and further increases performance on multi-core CPUs. Finally, since messages can be, and frequently are, overwritten on the channels, it means that fewer messages have to be processed, which should lead to even higher performance. On the channels, the oldest messages are overwritten first. If a new message arrives, all older message (of same type and from the same node) are obsolete anyway and there is no need to process them.

Besides the previously described channels, there is one more channel: the message exists channel, that goes between Sender and Processing. This channel is an auxiliary channel used by Sender to know when there exists messages to send. Whenever Processing sends a new message, a dummy value is sent on the message exists channel to notify the Sender thread. Section 3.3.3 describes in more detail how Sender uses this channel.

The last component of Figure 3.2 is Token and Count. Recall from Section 2.4 that Failure Detector monitors how many messages have been received from each node and also uses the piggybacked tokens. Since message reception (in Receiver) is separated from message processing (in Processing), and because messages and tokens can be overwritten, in our implementation, a special mechanism for the tokens had to be used. Token and count is a lock-free map that Receiver writes to. Whenever Receiver receives a message from a node, the latest token for that node is updated and the count (number of tokens received) from that node is incremented. Failure Detector then reads from Token and Count inside Processing. Suppose Failure Detector handles the token and count for node p_j. Instead of incrementing the heartbeats of all other nodes by 1, their heartbeats are increased by the count for p_j, as reported by Token and Count. Furthermore, only the last token is needed, since just like messages, earlier tokens are overwritten in the algorithm of [22].

3.3.1 FIFO and reliability

We use the common Internet jargon for reliable stream originated communications, which TCP/IP provides. That is, every message sent is eventually received and in the same order that it was sent. Moreover, UDP/IP does not offer such guarantees, but still, we assume communication fairness. That is, every message that is sent infinitely often is received infinitely often. We note that a violation of this assumption implies that eventually, no message from the sender ever arrives at the receiver.

All messages used in the algorithm of [22] require FIFO order. In our implementation, FIFO is achieved by letting the creator of a message (in Processing) attach a unique and monotonically increasing sequence number to each message it sends. Receiver keeps track of the highest sequence number h of each node, and discards all messages with a sequence number less than h.

Server messages do not need reliability, since only the latest version of a message is needed. Therefore, if some server messages are dropped, it poses no problems. Because the implementation periodically sends all server messages, if some messages are dropped, a retransmission happens soon anyway.

3.3.2 The processing thread

Algorithm 1: Pseudocode of the processing thread

```
1  local variables:
2  timeOfClientActivity := now();
3  while true begin
4      if pending client requests exist then
5          timeOfClientActivity := now();
6      if now() - timeOfClientActivity > 3 seconds then
7          wait 20 milliseconds;
8      viewEstablishmentIteration();
9      // Similar for the other modules
10 function viewEstablishmentIteration() begin
11     viewEstablishment.processReceivedMessages();
12     viewEstablishment.doForeverIteration();
13     viewEstablishment.sendMessages();
14 // sendMessages() calls are responsible for the monotonically increasing
     sequence numbers and for sending a dummy message on msg exists channel.
```

Algorithm 1 gives a high-level view of how Processing works. Processing is a `while true` loop that continuously runs one iteration of each of the three modules one at a time (lines 8-9). One iteration consists of first letting the module process received messages, then perform one iteration of its do forever loop and finally send its messages.

Lines 4-7 are of special interest. If there are no pending client requests, the system does not need to do anything. Without a special measure, Processing would run forever and consume unnecessary CPU time. Therefore, Processing keeps updating the time when there are pending client requests. If there are, Processing never sleeps. If there are no requests, Processing sleeps 20 milliseconds before performing one iteration. In this way, Processing runs fast when there are requests (in order to handle them quickly) and runs slowly when there are no requests (in order to save CPU time).

3.3.3 The sender thread

Algorithm 2: Pseudocode of the sender thread

```
15 local variables:
16 udpSocket; // Shared with Receiver
17 lastViewEstablishmentMessageSent[n];
18 timeOfLastViewEstablishmentMessage[n];
19 // Similar for the other message types
20 while true begin
21     wait until msg exists channel gets a message, then remove it;
22     viewEstablishmentMessageSending();
23     // Similar for the other message types.
24 function viewEstablishmentMessageSending() begin
25     for receiver in $[0, n-1]$ begin
26         let message := viewEstablishmentSendChannels[receiver].tryTake();
27         if shouldSendViewEstablishmentMessage(receiver, message) then
28             let bytes := message.serialize();
29             udpSocket.send(receiver, bytes);
30             lastViewEstablishmentMessageSent[receiver] := message;
31             timeOfLastViewEstablishmentMessage[receiver] := now();
32 function shouldSendViewEstablishmentMessage(receiver, message) begin
33     if $message \neq \perp$ then
34         if $message \neq lastViewEstablishmentMessageSent[receiver]$ then
35             return true;
36         let elapsed := now() - timeOfLastViewEstablishmentMessage[receiver];
37         if $elapsed > 100$ ms then
38             return true;
39     return false;
```

Algorithm 2 gives a high-level view of how Sender works. Sender consists of a `while true` loop that first waits until a dummy message arrives on the msg exists channel (line 21) and then proceeds to handle message sending for all modules (lines 22-23). Recall from Section 3.3 that a message is only sent on the msg exists channel if any of the modules has sent a message since the last iteration of Sender's `while true` loop. If Sender starts a new iteration but no messages are to be sent, Sender just sleeps on line 21 until a message to send exists. This saves CPU time when there is no activity.

Message sending (for View Establishment) is done in the function in lines 24-31. For each server *receiver*, a message is taken from the corresponding View Establishment send channel (line 26), see also Figure 3.2. If the message should be sent, it is serialized and then sent on the single UDP socket. The last view establishment sent to *receiver* and the time by which it happened are also updated. This last information is used to determine if a message should actually be sent or not, which happens in the function in lines 32-39. A message should only be sent if its non-⊥ and is different from the last message sent, or 100 ms has elapsed since a message was last sent to *receiver*. By only sending messages if they are different, the system avoids flooding the network with identical messages. The time out is used in case packet loss occurs.

3.3.4 The receiver thread

Algorithm 3: Pseudocode of the receiver thread

```
40 local variables:
41 udpSocket; // Shared with Sender
42 lastViewEstablishmentSeqReceived[n];
43 // Similar for the other message types.
44 while true begin
45     let bytes := udpSocket.receive();
46     let message := deserialize(bytes);
47     handleToken(message);
48     if $message.type = viewEstablishment$ then
49         handleViewEstablishmentMessage(message);
50     // Similar for the other message types.
51 function handleToken(message) begin
52     let sender := message.sender;
53     update last received token from sender;
54     increment count by one for sender;
55 function handleViewEstablishmentMessage(message) begin
56     let sender := message.sender;
57     let seq := message.seq;
58     if $seq > lastViewEstablishmentSeqRecevied[sender]$ then
59         lastViewEstablishmentSeqReceived[sender] := seq;
60         viewEstablishmentReceiveChannels[sender].send(message);
```

Algorithm 3 provides a high-level description of Receiver. Similarly to Processing and Sender, Receiver consists of a `while true` loop. The `while true` loop of Receiver blocks on a receive call on the UDP socket (line 45) and returns when a UDP message is received. The messages is then deserialized.

The first step after deserialization is to handle the piggybacked token. This happens in lines 51-54, where the last received token for the message's sender is updated (line

53) and the count is incremented by one (line 54). Incrementing the count means that one more message from this sender has been received. The two updates just mentioned happen to the Token and Count structure seen in Figure 3.2.

The second step after deserialization is to handle the message itself. Depending on the message type (line 48), the correct message handling function is called (lines 49-50). Message handling is done by checking if the message's sequence number is larger than the previously received sequence number for the sender and message type (line 58). If that is the case, the sequence number is updated (line 59), and the message is routed to the corresponding module through the correct receive channel (line 60).

3.4 Utility programs

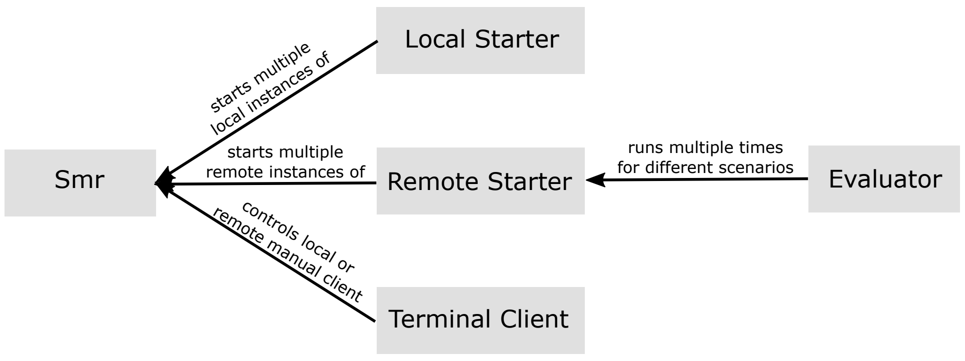
Local Starter
starts multiple local instances of
Smr
starts multiple remote instances of
Remote Starter
runs multiple times for different scenarios
Evaluator
controls local or remote manual client
Terminal Client

Figure 3.3: An illustration of the programs and how they relate to each other.

The implementation for this project contains a main program as well as several utility programs. The programs, shown in Figure 3.3, are *Smr*, *Local Starter*, *Remote Starter*, *Terminal Client* and *Evaluator*.

Smr is the main program, and is the actual implementation of the algorithm. The two previous sections, Section 3.2 and Section 3.3, have only been about Smr. The Smr program is stand-alone and can be launched using the appropriate command line arguments, which include a file with socket addresses of all hosts in the system as well as info regarding the parameters n, f and k.

Although it is possible to start multiple local instances of Smr manually, it is not very convenient. Starting and stopping multiple local instances of Smr, with different parameters and in different numbers each time, is very useful for debugging though. In order to make this debugging more streamlined, Local Starter created. With Local

Starter, a single command automatically starts the desired number of local Smr instances, and they are also automatically stopped when the user presses `ctrl+c`.

Remote Starter plays a similar role to Local Starter, but as its name suggests, the Smr instances are instead started on remote computers. These remote computers are accessed with SSH internally. Remote Starter takes care of everything that is needed to run Smr on the remote computers, including installation of Rust, upload of the source code, compilation and start and stop. To use Remote Starter, a file with socket addresses of the desired remote computers must be created. This file must also contain some extra information, such as which SSH keys to be used for login.

The client-side of Smr can be run in both automatic and manual mode. In automatic mode, all clients send requests all the time and immediately after they get $f + 1$ identical replies to their previous one. This mode is used for the performance evaluation, which is described in Chapter 4. In manual mode, the clients are instead possible to control manually for demonstration purposes. A client in manual mode starts a TCP server which Terminal Client should connect to. The user can then use Terminal Client to manually control both local and remote clients and interact with the SMR system.

The last program is Evaluator, which is used in the performance evaluation described in Chapter 4. Evaluator has two modes — *gather* and *aggregate*. In the gather mode, Evaluator uses a list of *scenarios*, runs one scenario at a time on remote computers using Remote Starter, collects the results from the computers and stores them in a file. A scenario is a combination of n, f and k for which the performance shall be investigated. Chapter 4 describes which scenarios are used in our evaluation. Since Evaluator and Remote Starter merely start one scenario at a time at multiple computers, they should have no effect on the actual results. In the aggregate mode, Evaluator reads multiple result files produced by the gather mode, and generates a Matlab file. This Matlab file contains all graphs that are shown in Chapter 4. The aggregate mode generates this file automatically so that it is easy to re-run the scenarios and to minimize manual work. In other words, no Matlab code is hand-written in this project.

We note that the utility programs do not have the same responsibilities as Cargo (Section 3.1). Cargo is responsible for building and testing, while the utility programs are responsible (among other things) for running multiple Smr instances.

3.5 New and reused code

Our implementation contains code that is entirely new for this project, but it also reuses previous code. The reused code comes entirely from previous open source[2]

[2]`https://github.com/osklunds/Distributed-SWMR-register`.

and Chalmers[3] projects done by us. The extent of the reuse varies from program to program (see Section 3.4). For Smr, the borrowed code is mainly the mediator setup (see Section 3.2). For Local Starter, Remote Starter and Evaluator, the basic structure of them is reused. However, we have made heavy changes to them, which includes general improvements as well as updating the tools to work for Smr. Terminal Client is entirely new in this project.

[3]Chibin: Course DAT085 in SP1 2019/2020 and course DAT295 in SP2 2019/2020. Oskar: Course DAT295 in SP2 2019/2020.

4 Evaluation

In order to answer the research questions posed in Section 1.3, we have conducted an experimental evaluation of our implementation of [22]. In the rest of this chapter, we present our evaluation criterion, the environments we use, the experiments we run, the results of the experiments as well as some conclusions.

4.1 Evaluation criterion

The evaluation criterion we consider is client latency. We define client latency as the time it takes from when a client sends a request to the SMR service until it gets $f + 1$ identical replies, shown in detail in Figure 4.1. $f + 1$ identical replies are needed in order to tolerate Byzantine faults, and is described more in [22]. Client latency is an important metric since it tells how fast the system is from the clients' perspective, the entities that are the actual users of the service.

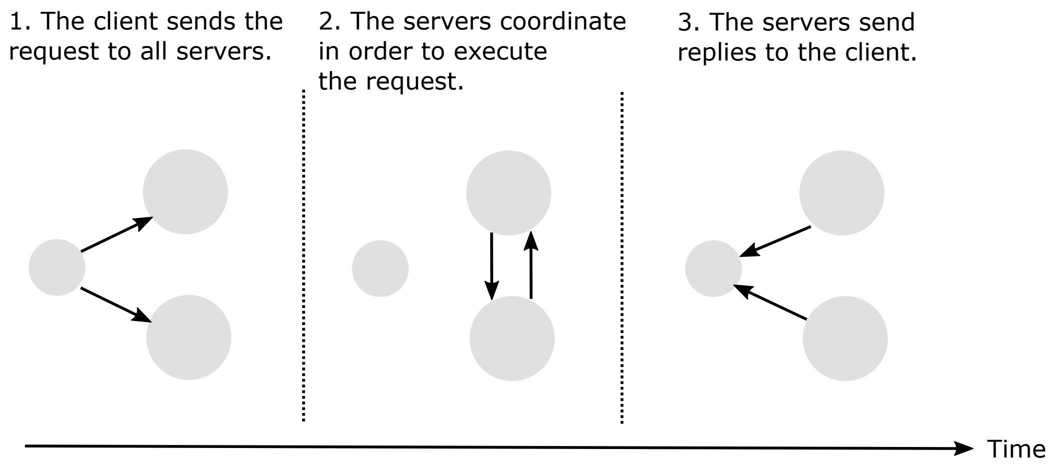
1. The client sends the request to all servers.
2. The servers coordinate in order to execute the request.
3. The servers send replies to the client.
Time

Figure 4.1: A diagram of how client latency is defined and measured. The latency of a single request, i.e., the client latency, is the time between step 1 and step 3. The small circles represent the client and the large the servers.

A related metric to latency is throughput. We do not consider throughput directly. However, in our experiments, clients send a new request immediately after $f + 1$ identical replies have been received. This means that the throughput can be found from the corresponding latency by inverting it.

4.2 Environments

In this section we describe the two different environments we use in the performance evaluation.

4.2.1 PlanetLab

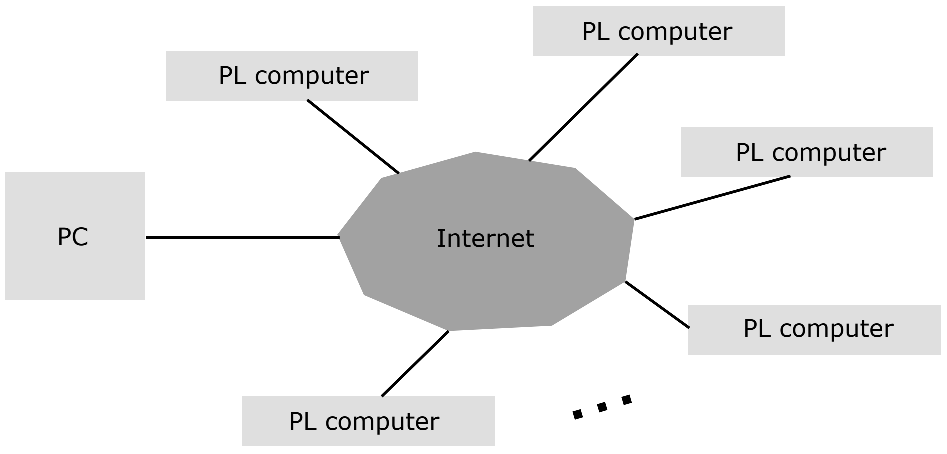

Figure 4.2: A schematic of PlanetLab.

PlanetLab [17] is an online testbed for distributed systems that consists of hundreds of computers connected to each other over the public Internet. A schematic of it is shown in Figure 4.2. We use PlanetLab Europe for our evaluation, which means that the computers used are located in various parts of Europe. A user of PlanetLab controls these computers through SSH, and it is possible to run arbitrary Linux programs on them. Figure 4.2 also contains a PC. This PC is our own local computer and is only used to control the system and collect data. The PlanetLab computers we selected were based on the computers the tool plcli [68] reported as working.

PlanetLab is a good platform for evaluating distributed systems in real world scenarios. It provides harsh conditions, including long and variable latency, packet loss, packet reordering and congestion.

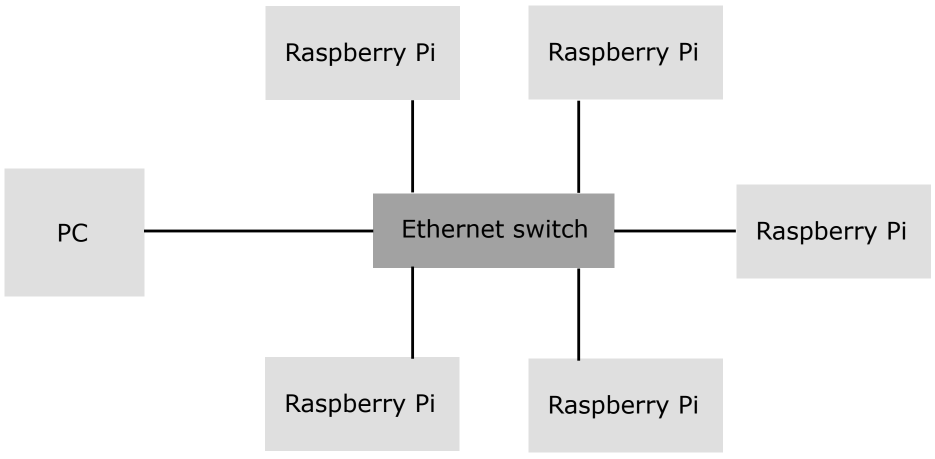

Figure 4.3: A schematic of the embedded evaluation environment.

4.2.2 Embedded

Our second environment is an embedded evaluation platform based on Raspberry Pis. A schematic of it is shown in Figure 4.3. We see that it consists of five Raspberry Pis and a PC connected by an Ethernet switch. The Raspberry Pis are model 4B with 4 GB of RAM, the switch is a Netgear ProSafe GS108 and all devices and cables support Gigabit Ethernet. Four of the Raspberry Pis are designated as server nodes, and run the server-side of the algorithm in [22]. The fifth Raspberry Pi is designated as a client node, and runs the client-side. Just like for PlanetLab, the PC here is not part of the actual system, but merely controls the system and collects the results.

The Raspberry Pi that acts as the client node can, depending on the experiment, run multiple client nodes. This is to be able to test scalability with many clients. While this node gets a higher load, the client-side of the algorithm is very light-weight compared to the server-side, which means that the results should not be affected too much by running multiple clients on a single physical node.

This embedded evaluation platform is used because it resembles an in-truck network better than PlanetLab does. With this platform, we are able to see how the algorithm would perform in such an environment, thus revealing the feasibility of using SMR to provide fault-tolerant services for critical functionalities in a trucks.

4.3 Experiments

We have conducted the following experiments in the evaluation.

1. **Scalability with respect to servers.** It is important to see how the system

scales when the number of servers increases. More servers lead to higher fault-tolerance, but might come at the cost of performance. The experiment is conducted with a varying number of servers. For PlanetLab, $n \in [1, 14]$, and for the embedded environment, $n \in [1, 4]$. A single client is used, meaning $k = 1$.

2. **Scalability with respect to clients.** It is important to see how the system scales when the number of clients increases. More clients has the potential to slow down the system, and it is useful to see how large the impact is. The experiment is conducted with four servers, $n = 4$. For both PlanetLab and the embedded environment, $k \in [1, 10]$. However, for the embedded environment, a single Raspberry Pi runs all the clients, as mentioned in Section 4.2.2.

In both experiments, we use $f = 0$ (because it is harder to offer performance predictions in the presence of failures) and let all clients perform 100 requests, after which the experiment is over. Recall from Section 2.4 that the algorithm of [22] adds every completed request to the local state and that the local state is sent in every Replication message. Due to this, only about 1,700 client requests can ever be completed by all clients in total before exceeding 65 KB, the maximum size of a UDP segment using fragmentation. Therefore, we chose 100 requests to allow for multiple clients in the experiments. We chose not to switch to TCP in our implementation due to an impossibility result [39] concerning reliable communication (which TCP provides), quorum systems (which [22] is) and self-stabilization.

The state machine used in the experiments is a dictionary, and the requests the clients issue are `INSERT` operations. The reason for this choice is to have a simple state machine that affects the performance as little as possible, so that the dominating performance factor instead is the SMR algorithm itself.

4.4 A lower bound on client latency

Before presenting the results, we provide here a lower bound on client latency. By keeping this bound in mind, we can compare how close to the optimal performance the implementation comes. In order to present a simple bound, let us assume that the *link latency* between every pair of nodes is the same. We define link latency as the one-way latency over a link from one host to another. Note that the link latency between two hosts is exactly half of the more commonly used round trip time. The bound is derived as follows.

Lower bound on client latency when $n = 1$. When $n = 1$, the system is not really a distributed system and cannot tolerate any failures. But for the sake of completeness we also analyze this case, which represents the classic client-server model [72, Chapter 2.3]. In this case, only two trips are needed, namely the client request to server and the server's reply back to the client. All server-side work of [22] happens completely locally on the single server of the system. Therefore the lower bound is two times the link latency.

Lower bound on client latency when $n \geq 3$. We analyze the case where $n \geq 3$ before $n = 2$ because the latter is a special case of the former. The message trips of Replication are analyzed as follows.

1. The client sends its request $\mathcal{R}$ to all servers, including the primary.
2. The primary assigns a sequence number to $\mathcal{R}$ and sets its status to `PREP`. The primary then sends this accepted request to all servers.
3. All backups see that $\mathcal{R}$ has been given a sequence number and hence add it to $reqQ$, followed by broadcasting this update to all other servers.
4. All servers see that $\mathcal{R}$ belongs to `knownReqs(PREP)` and change its status to `COMMIT`, followed by broadcasting this update to all other servers.
5. All servers see that all other servers consider $\mathcal{R}$ to have status `COMMIT`, so they execute it and send a reply to the client.

From the analysis we can see that at least five trips are needed for a client to get a reply. Therefore the lower bound in this case is five times the link latency.

Lower bound on client latency when $n = 2$. The analysis of this scenario is mostly the same as $n \geq 3$, and only differs in step 3 and 4. Since there are only 2 servers in the system, when the backup receives the message sent by the primary in step 2, the backup already has enough information to change the status of $\mathcal{R}$ to `COMMIT`. In other words, after the backup performs step 3, it can immediately perform step 4 without having to wait for other servers to send their updated states after they performed step 3, and one less trip is needed. Therefore, the lower bound is four times the link latency.

Based on the analysis above and the link latency of the network, we can roughly calculate the lower bound of client latency. The link latency in the two environments was measured using the `ping` command, by pinging five times from every host to every other host, and then calculating the average of all measurements. The average link latency was found to be 12.81 ms for PlanetLab and 0.083 ms for the embedded environment. It is worth noting that it is possible that the actual client latency is slightly smaller than the estimated lower bound, especially when n is small. This is because we use $f = 0$, which implies that the client only needs to wait for one reply. This means that for the last trip, only the fastest link to the client from the servers contribute to the overall client latency. We also note that the measurement was done at a single time, and not before each experiment. Since the PlanetLab computers have fluctuating speeds, the numerical values can differ slightly.

4.5 Number processing

In this section we describe how the raw measured numbers have been processed before they are presented. Both experiments from Section 4.3 were run 10 times each and the latency of each of the 100 requests was recorded individually. The node-ids of all PlanetLab computers were randomized between the runs. This was done since the various PlanetLab computers have different performance. These runs were then aggregated in the following manner, done for each request number individually. First the average latency of the x:th request was taken for all clients. This was repeated for all 10 runs individually. Then, for the x:th request for all runs, the two highest and the two lowest latencies were removed, and the average of the 6 remaining latencies was taken. This was done in order to mitigate the effect of outliers.

Depending on the graph, additional processing to the above may have been performed. To distinguish this, we use the two terms *measured latencies* and *latencies according to linear regression*. The former have received no additional processing. For the latter, linear regression has been performed in order to fit the measured latencies to a straight line. The term “latencies according to linear regression” thus refers to the latencies that the straight line predicts. When the linear regression was performed, the x values used were the x:th request, i.e. ranging from 1 to 100. The y values used were the measured latency of the corresponding x:th request, for a fixed number of servers and clients. In order to get a clearer understanding of this processing, we suggest to look at Figure 4.8, which is a good visualization of this processing.

4.6 Results

In this section we present the evaluation results of the two experiments described in Section 4.3.

4.6.1 Experiment 1: Scalability with respect to number of servers

We begin presenting the results of Experiment 1 with the contour graphs in Figure 4.4 and Figure 4.5, which show the results on PlanetLab and the embedded environment, respectively. These figures provide a good overview of the behavior of the algorithm of [22]. In both figures, the x-axis shows the x:th request and the y-axis shows the number of servers. The colored lines with numbers show the latency in milliseconds. We see that in both environments, there are two factors that determine the client latency: the number of servers and the number of completed requests. The higher any of these two numbers is, the higher the client latency. In the following paragraphs, we present these two major factors, as well as the numerical

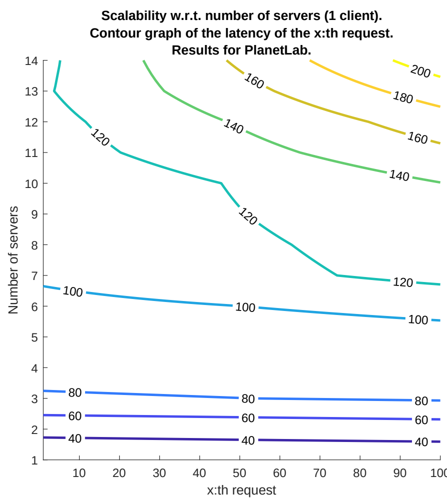

Figure 4.4: A contour graph of the result of Experiment 1 on PlanetLab. The x-axis is the x:th request. The y-axis is the number of servers. The numerical values shown on the colored lines are the latencies in milliseconds, according to linear regression of the measured latencies.

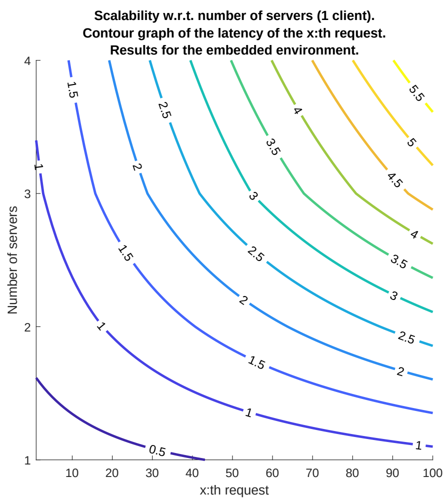

Figure 4.5: The graph corresponding to Figure 4.4, but shows the results on the embedded environment instead.

client latencies, in more detail.

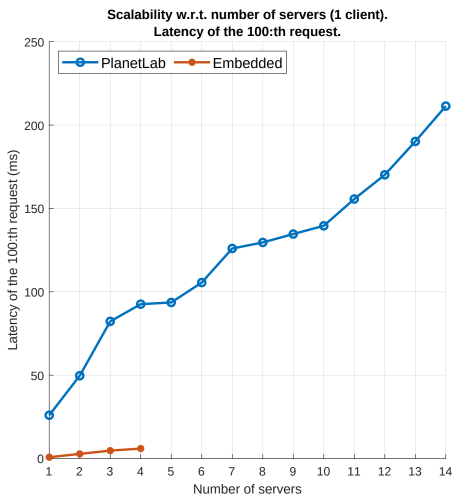

Figure 4.6: The result of Experiment 1. The x-axis is the number of servers and the y-axis is the latency of the 100:th request, according to linear regression of the measured latencies.

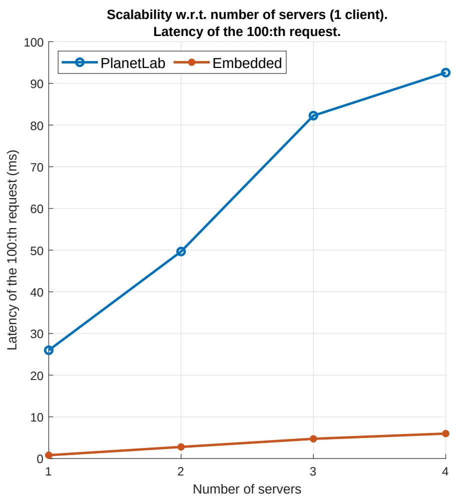

Figure 4.7: A zoomed version of Figure 4.6 that only shows between 1 and 4 servers.

Numerical client latencies. Let us start comparing the numerical client latencies. To this end, we turn to Figure 4.6 and Figure 4.7, which show how the client latency of the 100:th[1] request changes with the number of servers. We observe for PlanetLab that:

- The client latency changes from 26.0 ms to 92.6 ms as n changes from 1 to 4 (recall that the link latency of PlanetLab is 12.81 ms).
- This means that the client latency is 3.6 times as large for $n = 4$ compared to $n = 1$.
- It also means that the client latency is between 2.0 and 7.2 times longer than the link latency.

We observe for the embedded environment that:

- The client latency changes from 0.8 ms to 6.0 ms as n changes from 1 to 4 (recall that the link latency of the embedded environment is 0.083 ms).
- This means that the client latency is 7.5 times as large for $n = 4$ compared to $n = 1$.
- It also means that the client latency is between 9.6 and 72.3 times as long compared to the link latency.

By comparing the numbers across the environments, we see that increasing the number of servers has a bigger relative impact on the client latency on the embedded environment than on PlanetLab. We also see that the client latency has a bigger relative increase from the link latency on the embedded environment compared to PlanetLab. The reason behind these two observations is that client latency has two sources: link latency *and* local processing time. The link latency of the embedded environment is much lower than on PlanetLab, and therefore it is reasonable that as the work load on the CPU increases (more servers), this extra local processing time has a bigger relative impact on the client latency there than on PlanetLab. Furthermore, the Raspberry Pis have less powerful CPUs than the desktop computers of PlanetLab, which should make the higher work load even more noticeable.

Number of servers. Now we turn to the first factor which impacts the client latency, namely, the number of servers. Figure 4.6 and Figure 4.7 show how the client latency of the 100:th request changes with the number of servers, and we observe that it appears to increase linearly with the number of servers. The increase for $n \in [1, 3]$ is mainly due to the increased number of message trips needed as described in Section 4.4. When $n \geq 3$ the increase is mainly due to the fact that a majority of the work performed by Replication's do forever loop is linear in the number of servers. This work includes sending/receiving its local state to/from every other server as well as going through all servers' queues of requests and other local state. Since a majority of the server-side work grows linearly in the number of servers, it is reasonable that also the client latency does.

[1]We choose the 100:th request as an aggregated metric since the client latency grows with the number of completed requests.

We note that there is some work which grows faster than linear. For example, since every server sends messages to every other server, the total number of messages sent on the network increases quadratically. However, it appears that on both PlanetLab and the embedded environment, the network is much faster than the hosts themselves, and can handle the quadratic increase in load without becoming the bottleneck of the system. Therefore the graphs do not show any quadratic tendencies. On the other hand, if even more servers are used, the client latency might start to increase quadratically as the network becomes more congested. Replication also contains some sequential logic that is exponential in the number of servers. With only a few servers, this sequential logic appears not to contribute significantly to the local processing time. However, just like with the network, as many more servers are used, it is likely that this will start having an effect.

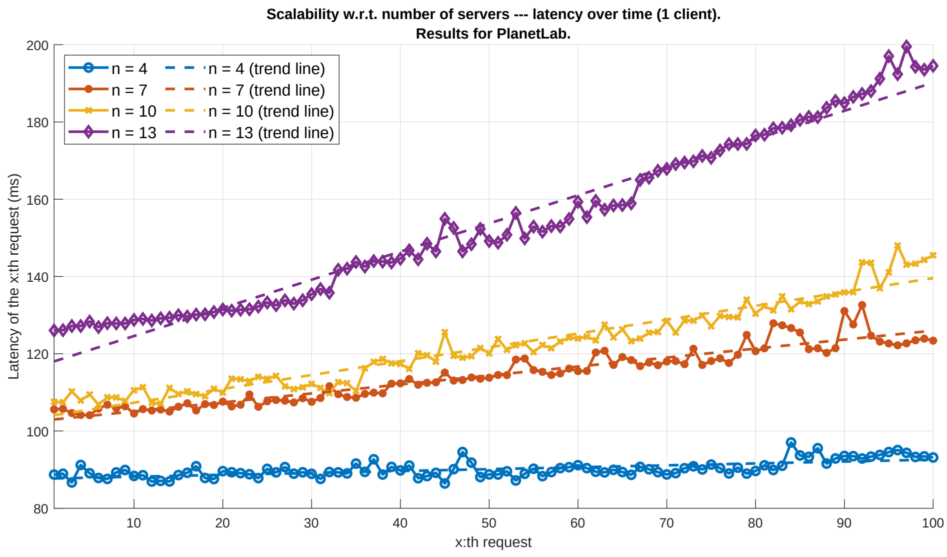

Figure 4.8: The result of Experiment 1 on PlanetLab. The x-axis is the x:th request. The y-axis is the latency in milliseconds. The bold lines are the measured latencies while the dashed lines are the corresponding trend lines, according to linear regression.

Number of completed requests. The second factor which impacts the client latency is the number of requests that have been completed. Figure 4.8 and Figure 4.9 show how the client latency changes as more requests are completed. We observe that in both environments and for all number of servers, the client latency increases linearly as more requests are completed. Recall from Section 2.4 that in the algorithm of [22], each completed request is added to the local state, and is then never removed. Furthermore, in every Replication message sent, the entire local state is sent. Since the server-side work increases linearly with the number of completed requests, it is reasonable that the client latency also does so.

Note that for all number of servers, and for both environments, there is a small peak at approximately 45 completed requests, confer Figure 4.8 and Figure 4.9. It is interesting that the peak appears consistently across these many cases, yet,

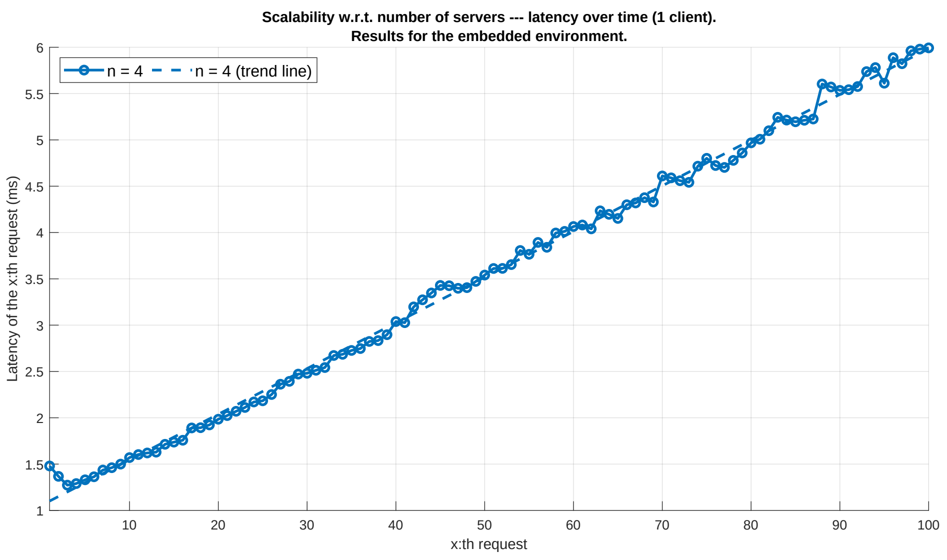

Figure 4.9: The result of Experiment 1 on the embedded environment. The x-axis is the x:th request. The y-axis is the latency in milliseconds. The bold lines are the measured latencies while the dashed lines are the corresponding trend lines, according to linear regression.

there is nothing in the algorithm that would suggest that this peak should appear. However, the peak coincides when fragmentation starts being used. Recall from Section 4.3 that about 1,700 requests fit in a single fragmented UDP datagram of length 65,536 B, which means that each request contributes about 38.6 B to the messages. Note that $38.6 \cdot 45 = 1,700$. This is almost 1,500 B, the length at which UDP datagrams have to be fragmented over Ethernet. It might be the case the fragmentation starts being used at 45 completed requests. Since it is a peak, and not a permanent increase, this explanation is not fully satisfactory, but it is an interesting connection.

Comparison to the lower bound. Before leaving Experiment 1, let us compare the results with the lower bound on the client latency of Section 4.4. For PlanetLab and $n \in [1, 2]$, the bound is 25.62 ms and 51.25 ms respectively. In Figure 4.7, we see that the measured latencies match this bound very closely, although for $n = 2$, the measured latency is slightly below the bound. However, recall that the bound is only approximate and that the link latency is an average. It may be that the servers used in this experiment by chance tended to be faster than average. For $n \geq 3$, the lower bound on the client latency is 64.05 ms. By examining Figure 4.6 and 4.7, we see that the measured latencies are indeed greater than the lower bound. It is also worth noting that the client latency for $n \leq 6$ is less than 120 ms, only two times higher than the lower bound. This means that the performance is of the same magnitude as the optimal latency, meaning the algorithm and implementation fare relatively well. As for the embedded environment and $n \geq 3$, the client latency is well above the lower bound of 0.42 ms. This is because the link latency is very low and the local processing time is the bottleneck here, due to the Raspberry Pi's lower processing power.

4.6.2 Experiment 2: Scalability with respect to number of clients

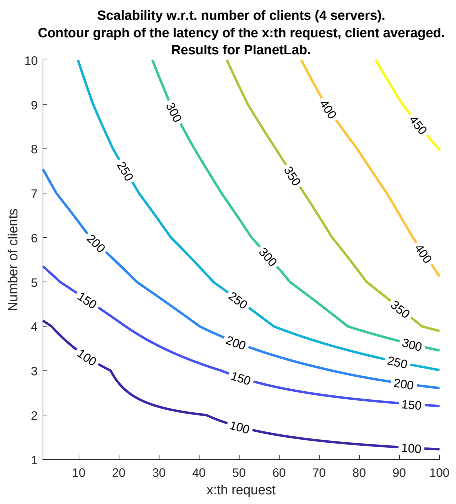

Figure 4.10: A contour graph of the result of Experiment 2 on PlanetLab. The x-axis is the x:th request. The y-axis is the number of servers. The numerical values shown on the colored lines are the latencies in milliseconds, according to linear regression of the measured latencies.

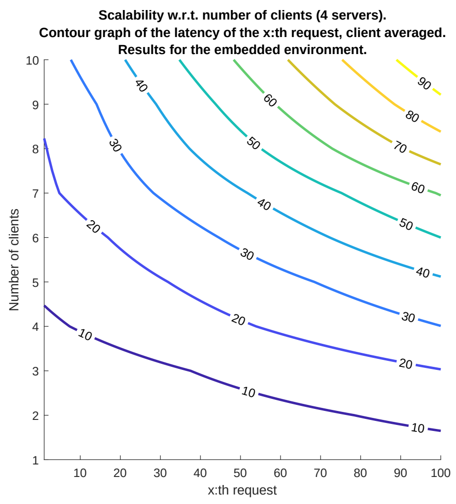

Figure 4.11: The graph corresponding to Figure 4.11, but shows the results on the embedded environment instead.

Similarly to Experiment 1, we begin presenting the results of Experiment 2 with contour graphs, which can be found in Figure 4.10 and Figure 4.11, for PlanetLab and the embedded environment, respectively. The x-axis is the x:th request, the y-axis is the number of clients and the colored lines with numbers show the client latency in milliseconds. Just like in Experiment 1, we observe that the client latency becomes longer as more requests are completed. However, we also see that more clients also lead to longer client latencies. From the figures, we see that the client latency on PlanetLab is between 100 and 450 ms (the link latency on PlanetLab is 12.81 ms) and that the client latency on the embedded platform is between 10 and 90 ms (the link latency of the embedded platform is 0.083 ms). In the following paragraphs, we present the results in more details.

Number of clients. Figure 4.12 shows how the client latency of the 100:th request changes when the number of clients changes. We expect that the client latency increases linearly with the number of clients, because with twice as many clients, twice as many requests are performed and thus the local state grows twice as fast. By a similar argument as for Experiment 1, a doubled local state should lead to a doubled client latency as well. In Figure 4.12, we see that our expectations

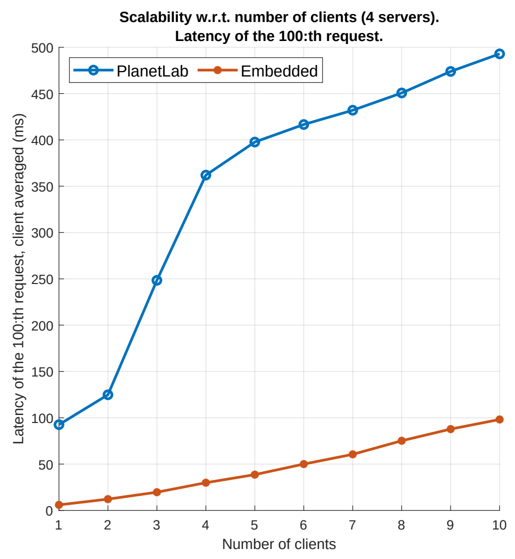

Figure 4.12: The result of Experiment 2. The x-axis is the number of clients and the y-axis is the latency of the 100:th request, according to linear regression of the measured latencies.

hold very well for the embedded environment, and for PlanetLab for $k \geq 4$, but not as well for $k < 4$. This unexpected behavior can be due to similar reasons as those given in the next three paragraphs.

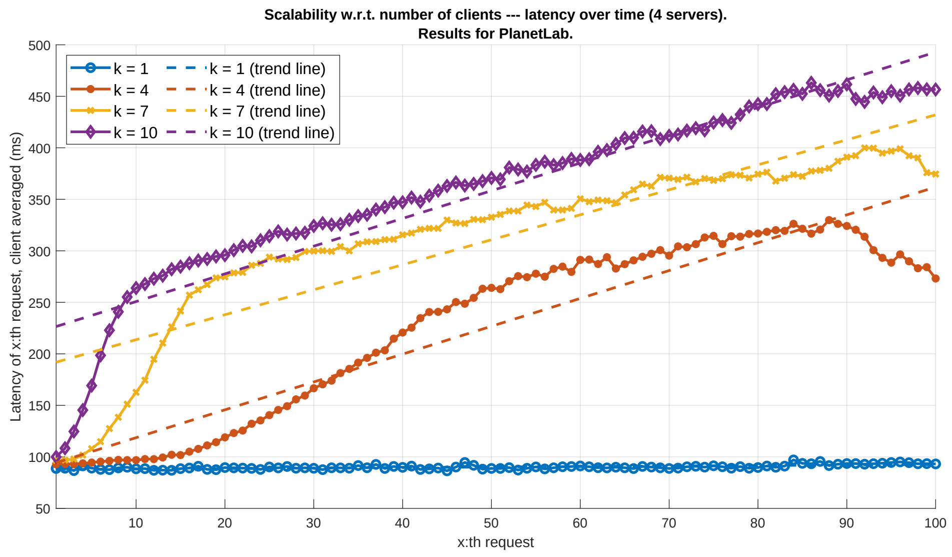

Figure 4.13: The result of Experiment 2 on PlanetLab. The x-axis is the x:th request. The y-axis is the latency in milliseconds. The bold lines are the measured latencies while the dashed lines are the corresponding trend lines, according to linear regression.

Number of completed requests. Figure 4.13 and Figure 4.14 show how the client latency changes as more requests are completed. Recall from Section 4.6.1

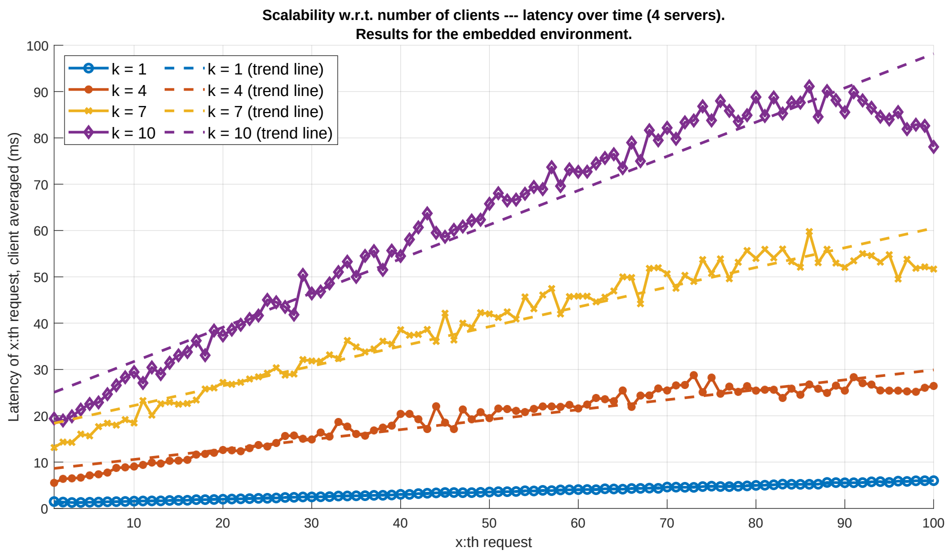

Figure 4.14: The result of Experiment 2 on the embedded environment. The x-axis is the x:th request. The y-axis is the latency in milliseconds. The bold lines are the measured latencies while the dashed lines are the corresponding trend lines, according to linear regression.

that we expect the client latency to increase linearly with the number of completed requests. In both environments, this expectation is matched well for $k = 1$ client. For larger k, this expectation is still matched somewhat well in the embedded environment.

In Figure 4.13 we see that PlanetLab for $k > 1$ does not match the expectations as well as the embedded environment, especially in the beginning and the end. A possible reason could be that all clients perform 100 requests and then stop, but their start and stop times might be slightly out of sync due to the varying speeds of the PlanetLab computers. In the beginning, not all clients are active because some are slower and have not started yet, and in the end, not all clients are active either because some clients have finished earlier. The slowest case is when all clients are active, because then, the Replication messages are the largest due to many pending requests, and we know from Section 4.6.1 that larger Replication messages lead to longer a client latency. Furthermore, when all clients are active, it also increases load and congestion on the network, which could further increase the client latency. Since the slowest case happens in the middle of the experiment, this could explain why the client latency is higher between the 30:th and 80:th request, and lower before 30:th and after the 80:th.

The phenomena just described is more visible on PlanetLab (Figure 4.13) than on the embedded environment (Figure 4.14). This could be because the computers in the embedded environment are identical, while on PlanetLab they are much more heterogeneous. It could also be that the network in the embedded environment is much more powerful in relation to the computers than the corresponding PlanetLab network and computers.

4.7 Conclusions

Our first research question asks how the performance changes as more requests are completed. From both Experiment 1 and Experiment 2, we see that the client latency increases linearly with the number of completed requests, with some deviations. The linear increase is expected because each completed request is added to all servers' local state, and the entire local state is sent in every server-side message. The deviations from the expectation are possibly due to the fact the clients start and finish slightly out of sync.

Our second research question asks how well the system scales with the number of servers and clients. We expect that the client latency should increase linearly with the number of servers, since the server-side work, such as sending/receiving message, increases linearly with the number of servers. From Experiment 1, we see that the results match this expectation very well. We expect that the client latency should increase linearly with the number of clients as well. This is because more clients also increases the sever-side work linearly, in terms of a longer list of completed requests. From Experiment 2, we see that the results match this expectation very well for the embedded environment, and for PlanetLab with more than four clients. There are however some deviations for PlanetLab with less than four servers.

Our third research question asks how the system performs in different environments. We performed our experiments in two environments: PlanetLab and an embedded network consisting of Raspberry Pis. We found that the overall behavior is the same in both environments, but that the client latencies are significantly lower in the embedded environment. These results are expected since the embedded environment has a much lower link latency than PlanetLab.

5 Applications in trucks

In this chapter we present three possible use cases of SMR in trucks: Diverse Redundancy, fault tolerant logging, and distributed services.

5.1 ISO 26262: Diverse Redundancy

In this section, we share an idea of how SMR can be used to implement Diverse Redundancy for ECU software. We provide a background on safety in trucks, how SMR and Diverse Redundancy are related, what SMR could look like in a truck and some of the challenges of using SMR in a truck.

5.1.1 Background on safety in trucks and ISO 26262

It is important that vehicles on the road, such as trucks, behave in a safe way. Due to their high speeds and heavy weights, unintended behavior can clearly lead to severe consequences. Therefore, so called *safety goals* are identified, examples of which can be found in Table 5.1. A violation of these safety goals can occur due to several reasons, including problems at the electric and software levels.

Table 5.1: Three examples of safety goals identified for trucks.

Unintended activation of adaptive cruise control shall not occur.
Engine brake shall not be applied when not requested.
The transmission shall not engage a reverse gear when driving at high speed in forward direction.

Related to trucks and safety is the ISO 26262 standard [1]. ISO 26262 is a standard on functional safety of electric and electronic systems of automotive vehicles. It contains safety requirements and recommendations of methods to increase the safety of electric and electronic systems. One of these methods related to software is *Diverse Redundancy* [3] (or *Independent Parallel Redundancy* [2]), which means that a single piece of software is replaced by multiple versions of the same software. These multiple versions should be developed by different teams and be run on different hardware so that the same bugs do not appear in all of them. The advantage of this scheme is that a bug in one software version cannot bring the whole system down — as long as the same bug does not occur in the other versions, the system can still continue to function.

5.1.2 State machine replication for Diverse Redundancy

State machine replication is a method which can facilitate reaching the safety goals in a way that ISO 26262 suggests, namely, SMR is a way to provide the aforementioned Diverse Redundancy. Recall from Section 1.2 and Section 2.2 that SMR removes a single point of failure. This means that if the single software instance crashes due to a bug, or the CPU running it stops working, the system as a whole can still work. Furthermore, the SMR algorithm of [22], which this project has focused on is also self-stabilizing and Byzantine fault tolerant (see Section 2.3). In particular, this means that not only crashing software can be handled. Also software that keeps running, but produces incorrect results, can be handled. These incorrect results can be due to the program not following the specification, bugs such as buffer overflows or malicious attacks.

Diverse Redundancy, also known as N-version programming in the literature [16], has previously been used successfully for NFS (Network File System) [15]. The algorithm used in that case was a Byzantine fault tolerant SMR algorithm, although it was not self-stabilizing. When comparing the performance between standard unreplicated NFS, and NFS replicated using SMR, the latter was found to be only 3 % slower. This shows that Diverse Redundancy can indeed be practical in real world systems. Furthermore, by using a technique called Abstract Base Encapsulation, multiple versions of the same software can be easier to combine [71].

5.1.3 What state machine replication could look like in trucks

One option of using SMR for Diverse Redundancy inside a truck is to let the entire SMR system reside inside a single multi-CPU ECU, as depicted in Figure 5.1. By having one SMR instance per CPU, each running one version of the ECU's main software, the likelihood that a software bug causes the ECU to malfunction is reduced.

In order to use the above scheme, one major problem has to be solved, namely, the ECU software must be adapted to the client-server model. Recall from Section 1.2 that the program replicated with SMR must be based on clients that send requests and servers that act on these requests and send replies. This is quite restrictive and current ECU software is most likely not written in this way. However, it might be possible to adapt ECU software into an SMR-friendly client-server based version as depicted in Figure 5.2.

The ECU software is decomposed into one server-side and multiple client-sides. The server-side contains all existing ECU logic, such as data structures with vehicle state, how to select the correct gear, what actions should be taken based on incoming

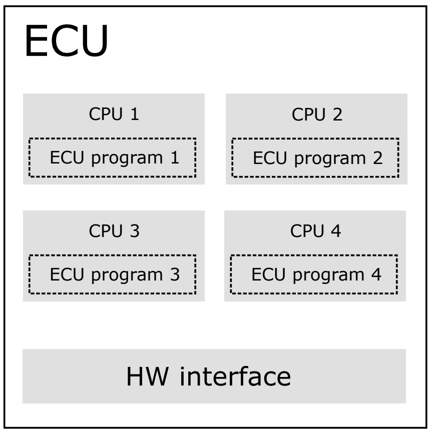

Figure 5.1: SMR used inside an ECU. All four ECU programs do the same thing and follow the same specification, but they are developed by different teams and run on different CPUs. They are coordinated by SMR.

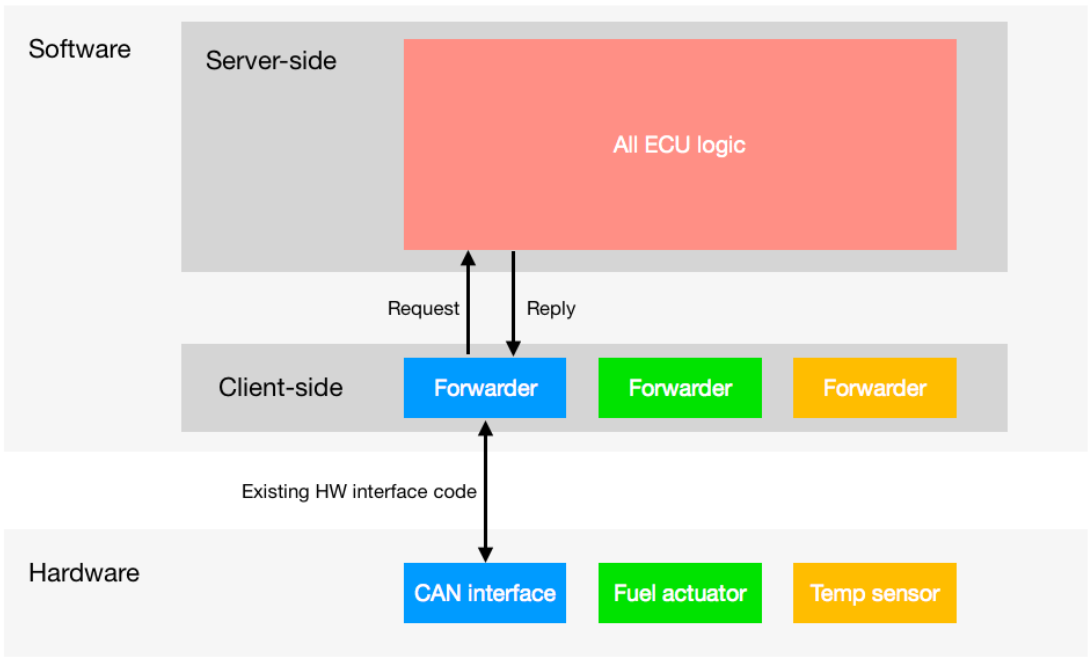

Figure 5.2: A schematic of ECU software adapted to the client-server model. The server-side contains all ECU logic. The client-sides only contain hardware interfacing code and act as forwarders between the ECU logic and the hardware.

CAN messages, etc. Of the existing ECU software, as much as possible should be decomposed into the server-side, since only the server-side is replicated by SMR. The client-sides should be simple forwarders. For example, when a CAN message arrives, the client-side simply issues a request, containing this CAN message, to the server-side and lets the server-side decide what should be done. The client-sides should be kept as small as possible since they are not replicated, and should only contain the hardware interface logic.

In addition to handling the hardware, the client-sides can also act as schedulers in the above scheme. Recall that in the client-server model, the servers only take actions based on client requests. Since the ECU logic might contain actions that should be performed periodically, these events must be triggered in some way. A possible solution is for the client-sides to periodically send requests whose purpose is to tell the server-side to perform one of those actions. Furthermore, this can also solve another problem. Based on the ECU logic, the server-side might want to tell the hardware to perform some action, or send a CAN message. These “requests” from the server-side can be sent in the replies to the periodic trigger requests, to inform the client-sides to do some particular action.

An overview of the ideas presented in the two previous paragraphs appears in Table 5.2.

Table 5.2: An overview of how incoming, periodic and outgoing events can be handled.

Event type	**Examples**	**How to handle**
Incoming	CAN message received, temperature measurement	The client-side sends a request upon the event.
Periodic	Set fuel/air ratio, read engine temperature	The client-side periodically sends a “schedule”-request, and performs the action the server-side includes in the reply.
Outgoing	CAN message to send, emergency brake	The server-side piggybacks these actions in the replies to the “schedule”-requests.

We note that the ECU logic might be large and complex. However, it is not necessary to use the above scheme for *all* ECU logic. Perhaps only the most critical logic needs to be replicated in the above way.

5.1.4 Problems

Using SMR inside an ECU has the advantage of increased fault tolerance, but it also comes with some problems which we describe here.

1. SMR algorithms are non-trivial, which means that the SMR service itself is a potential source of bugs if not implemented carefully, especially if it is responsible for running all other software of the ECUs. For safety critical systems, it might be more appropriate to have a simple but reliable backup system instead. One example could be to let the transmission disengage the clutch if the engine starts behaving in a bad way.

2. SMR has the potential to be slower and bring more overhead compared to an

unreplicated program. This is due to both the SMR algorithm's overhead itself and to the fact that the program has to be rewritten to be based on sending and receiving messages. Section 5.1.2 mentioned that NFS became only 3 % slower when replicated using SMR. However, NFS is already a networked system, and the network latency might be the bottleneck. The relative slowdown is possibly larger when using SMR within a single ECU.

3. ECU software might not be possible to rewrite into the client-server model. Furthermore, even if it is possible, it might be expensive to do so.

4. Due to the theoretical limits mentioned in Section 2.3, at least 4 versions of the same software is needed for Byzantine fault tolerance. Developing 4 versions of the ECU software is most likely very expensive.

5. The entire ECU software is not protected by Diverse Redundancy, only the server-side is. Software bugs in the hardware interfacing code are not masked by SMR.

5.1.5 Conclusions

SMR is a possible way to increase safety in trucks by reducing the possibility of software bugs. SMR is a way to implement Diverse Redundancy, which is suggested in ISO 26262. However, in order to use SMR for ECU software, the ECU software must be rewritten to fit the client-server model. There are also various problems with this approach, such as cost and performance.

5.2 Fault tolerant logging

If a truck is involved in an accident, it is interesting to have access to the logs generated by the truck. These logs include for example engine hours, total distance travelled and sensor data. It is important to store the logs in multiple physical locations of the truck, because if it is involved in an accident, there is a chance that some of the storage locations are destroyed.

With SMR, adding fault tolerant logging is relatively straightforward. The first step is to define what data structure to store the logs in. One possible data structure is a dictionary with ECU-id and timestamp as keys, and the log entry as values. Then the server-side of the SMR algorithm is run on a few ECUs distributed in various physical locations of the truck while the client-side is run on all ECUs that should send logs. This is shown in Figure 5.3. The server-side can be seen as a daemon that is run on the ECUs and the client-side can be seen as a library that can be called from the software running on the ECUs when it needs to log something.

It is possible to enrich the server-side with more features such as consolidation and stable storage, also shown in Figure 5.3. With consolidation, the logs are prevented from growing too large. For example, perhaps it is sufficient to only store hourly

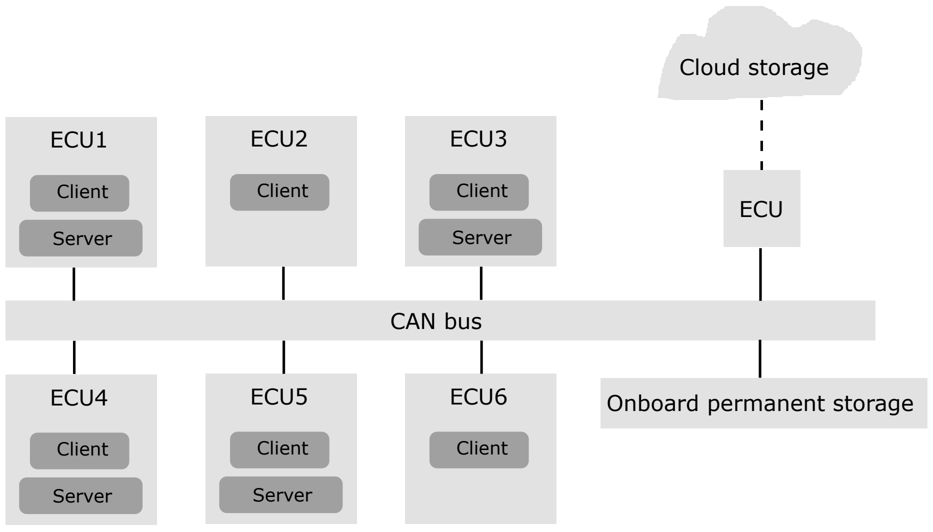

Figure 5.3: A schematic of an ECU network and fault tolerant logging. Four of the ECUs run the server-side of SMR while all six run the client-side. The server-side and client-side run in parallel with all other programs the ECUs run. The CAN bus might also be connected to cloud storage and onboard permanent storage.

versions of logs older than 24 engine hours. With stable storage, the logs can be moved from the ECUs into some other storage. One example could be cloud storage, to which the logs are sent over cellular network. Another example could be some form of onboard permanent storage, like a black box. Since SMR is not limited to simple programs, features such as consolidation and stable storage are also possible to include at the server-side. These features can be performed in the background in a way transparent to the client-side.

5.3 Distributed services

SMR can be used to implement fault tolerant distributed services in the CAN network of a truck. The ECUs would all be part of a distributed system where they collaborate on some shared tasks. Since SMR is general, there are many possible services that can be implemented.

We note that in current ECU network architectures, the aim is to let the ECUs be self-contained and independent. Of course, there is still communication between them, e.g. the current gear is sent from the transmission ECU to the engine ECU. However, despite this communication, the ECUs do not collaborate on tasks. The suggestions we make imply that the ECUs would collaborate a lot more and be less self-contained, which is different from current architectures. However, these suggestions might still be useful for potential future ECU network architectures where more collaboration is used.

5.3.1 Shared memory

One service that could be interesting is high-level shared storage in the form of shared memory [5, 19, 52], which can make coordination between the ECUs easier. ECUs and shared memory is depicted in Figure 5.4. Shared memory enables ECUs to store data that multiple of them need. One example of such data could be the current gear, which the transmission ECU knows. Instead of having the transmission ECU send the current gear to all ECUs that need it, the transmission ECU can write the current gear to a shared memory location allocated to the current gear, and then all ECUs that might possibly need it can read from that memory location.

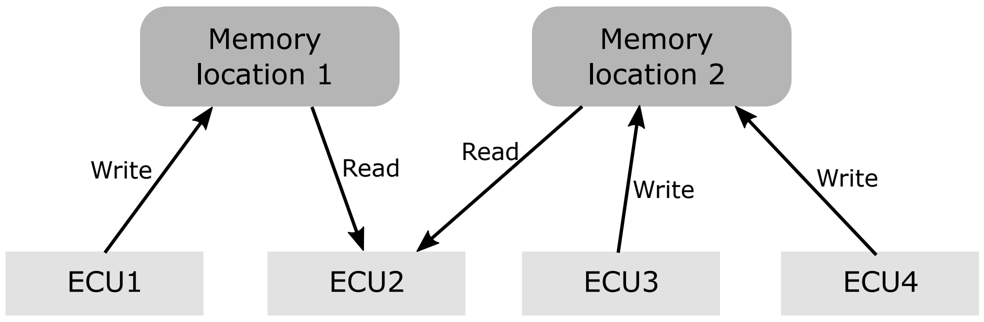

Figure 5.4: An illustration of four ECUs and two shared memory locations. All ECUs can read and write to the shared memory.

There are several possible advantages that shared memory can bring to ECU networks, compared to sending the data directly in CAN messages over the network. First (continuing with the previous example), the transmission ECU does not need to be aware of the ECUs that need the current gear. All ECUs that need the current gear will read it on their own, with no additional action needed to be taken by the transmission ECU. Second, shared memory can bring additional consistency guarantees, such as *linearizability* [54], compared to sending data directly in messages over the network. For example, linearizability implies that if two ECUs read the shared memory, the second read will return a value that is *at least as recent* as the first read. Third, multiple ECUs can write to the same memory location [10, 39]. The shared memory algorithm makes sure that there is no disagreement on which value should be used, even if multiple ECUs write to the location at the same time.

5.3.2 Safety kernel

A safety related service is the *safety kernel* [11, 12, 13, 14]. The safety kernel is a central architectural component that makes decisions on what the current Level of Service (LoS) should be. When every component of the truck functions as expected, the LoS is “normal”. The LoS can degrade when failures are detected in the truck.

For example, if a failure is detected in one of the brakes, the LoS can degrade to “limp home”, where the maximum speed is limited, so that the truck can still move in a safe way in order to reach a workshop where its brake can be fixed. If all brakes fail, perhaps the LoS will be ‘no op’. It is useful to have a LoS such as “limp home” so that there is no need to wait for a tow truck if the truck is still safe to drive, albeit only slowly. The safety kernel decides on the highest possible LoS that is safe to maintain. If a LoS is safe or not depends on many safety-related parameters such as the status of all the truck’s components, the weather and if there are any nearby hills. Since the safety kernel uses many inputs and its logic can be complex, it can be easier to design it as a centralized component, that resides in only a single logical location, as shown in Figure 5.5. However, by combining the safety kernel with SMR, it is still possible to have the safety kernel in multiple physical locations in order to have redundancy. In other words, SMR allows the design of complex components to be centralized, but still have the advantage of redundancy without extra effort at the design time.

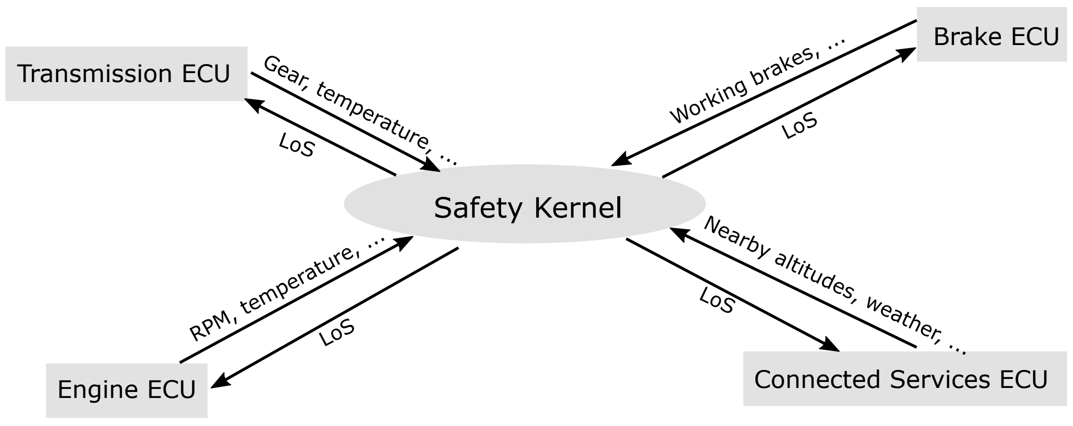

Figure 5.5: An illustration of the interaction between the safety kernel and ECUs of a truck. The safety kernel exists in one logical location and all ECUs send safety-related input data to it. Based on the input, the safety kernel decides on the highest safe LoS and sends this to all ECUs, which in turn take appropriate actions.

6 Global Reset

This chapter is about a technique called *Global Reset*, which can be used together with many algorithms, including the SMR algorithm of [22]. This is a technique for a global system reset for regaining consistency and enforcing garbage collection. Global Reset can be used to bound unbounded counters, buffers and message sizes used in algorithms such as [22]. Moreover, it can be used to bound leaking resources or any other kind of detectable consistency violation. For example, it might also be possible to use Global Reset to bound the unbounded growth of client latency that [22] exhibits, as discovered in Section 4.6.

In the context of self-stabilization it is important that all variables used are *bounded* [21, Chapter 2.8]. However, many (self-stabilizing) algorithms, including [5, 22, 38, 39, 52, 64], use *unbounded* integer variables, often as sequence numbers, counters or timestamps. By using 64-bit integers, the maximum value MAXINT that can be stored is in practice never reached [1]. However, we note that an arbitrary transient fault can corrupt all variables and make them assume values that are close to MAXINT, meaning an overflow can happen in practice. The problem outlined above is something all self-stabilizing algorithms have to deal with, and thus, it can be useful to create a general technique that can be used by many self-stabilizing algorithms. This is what Global Reset is.

Even though the Global Reset technique is not Byzantine fault tolerant, it can still be combined with [22]. *Dolev et al.* [22] assume that recovery from transient faults does not occur in the presence of Byzantine failures. Moreover, reaching MAXINT can only occur due to a transient fault, as we explain above. Thus, by proposing a self-stabilizing algorithm for safely resetting the system, we offer [22] the ability to reset the system when the system in [22] reached its maximum capacity with respect to the value in it counters, the messages stored in it buffers or the amount of information that it repeatedly communicates.

In this chapter, we provide a high-level description of the Global Reset technique. A more detailed description can be found in Appendix A. We also provide an ex-

[1]If incremented once every nanosecond, the time it takes to count from 0 to $2^{64} - 1$ is $\frac{2^{64}}{10^9} \approx 1.8 \cdot 10^{10}$ seconds ≈ 585 years. This is most likely longer than most computer systems will be used.

perimental performance evaluation of Global Reset in a similar vein as in Chapter 4.

6.1 High-level description of the Global Reset technique

How to bound an algorithm using the Global Reset technique of course depends on the algorithm and how it uses unbounded variables. However, one common usage pattern is that client operations cause the counters of the algorithm to increase. This is the case for the SMR algorithm by [22], where each client request increments the current sequence number by one. A similar pattern also appears in algorithms such as [5, 39, 52, 53]. For such cases, the Global Reset technique works as follows.

1. Once a server stores MAXINT in any variable, it stops responding to client operations.

2. The server calls the *global reset algorithm*, which in a delicate way resets the variable to 0 at all servers, while the remainder of the state is preserved. This other state includes for example the current state machine state in the case of [22].

3. When the global reset algorithm is done, the servers resume to normal execution and start serving clients again.

This was a high level overview of the Global Reset technique. Appendix A contains some more details on how to use it with MW-ABD, an algorithm for shared memory emulation. As for the global reset algorithm, we use the one presented by Georgiou *et al.* [51].

6.2 Implementation and Evaluation

In order to investigate how the Global Reset technique performs in practice, we have implemented the global reset algorithm by Georgiou *et al.* [51], which is the core algorithm of Global Reset. The implementation reuses the framework of Chapter 3 that we developed for the SMR algorithm, including Communicator and Mediator shown in Figure 3.1. The only difference is that Mediator's loop runs the global reset algorithm's do-forever iteration instead of the three modules of the SMR algorithm. To evaluate how long time it takes to perform a Global Reset, we conduct an evaluation of the global reset algorithm in a similar manner as for the SMR algorithm (see Chapter 4). In the following sections we present how the evaluation is carried out and the results obtained.

6.2.1 Evaluation criterion

The evaluation criterion we use is *global reset time*, which is defined as the time it takes from when a server first proposes a tag until it calls $localReset()$. It is worth noting that a possibly more accurate definition of Global reset time can be until the time that the last call to $localReset()$ among all servers returns. However, this is much harder to measure since it would require the servers to synchronize their real-time clocks to sub-millisecond precision. As the global reset operation happens on the server-side, we do not consider clients in this evaluation.

6.2.2 Experiment

As mentioned earlier, only the participation of servers is considered in this evaluation. Therefore our experiment focuses on the scalability with respect to the number of servers. Intuitively, when there are more servers, we expect to see an increase in the global reset time. This is because each server needs to communicate with more servers.

We conduct the experiment on both PlanetLab (Section 4.2.1) and the embedded environment (Section 4.2.2). On PlanetLab, we increase the number of servers from 2 to 15. On the embedded environment, we increase the number of servers from 2 to 5.

In the evaluation, we run the experiment 10 times and for each run we invoke the global reset operation 10 times. Before each run, the order of the PlanetLab hosts is randomized since they have different speeds and link latencies. In this way, for each number of servers, we have 100 samples of global reset time. From these samples we remove the 5 smallest and 5 largest, in order to mitigate the effect from outliers, and take the average of the rest.

6.2.3 Estimation of global reset time

Before presenting the results, we present an analytical estimation on the global reset time. Since there are no clients, we only analyze the case where $n \geq 2$, since a single node is not a distributed system. Please refer to [51] for the global reset algorithm itself. The estimation is derived as follows.

1. When node p_i calls $propose()$, it changes its own phase from 0 to 1. After one round trip of communication, where all nodes exchange information by sending and receiving, p_i notices that all nodes echo its phase, and thus sets the $all[i]$ flag to true.

2. After the next round trip of communication, all other nodes also echo that p_i's $all[i]$ flag is true. This means that p_i moves from phase 1 to phase 2. During this process, the $all[i]$ flag is set to false since p_i's state has changed.

3. After the next round trip of communication, all other nodes acknowledge p_i's

new phase, phase 2. Therefore the $all[i]$ flag is set to $true$ again.

4. After the next round trip of communication, similarly to step 2, all other nodes acknowledge p_i's state, including $all[i] = true$. This means that p_i calls $localReset()$.

From the above analysis, we see that it takes 4 communication round trips from when $propose()$ is called until $localReset()$ is called. Recall from Section 4.4 that the average RTT is 25.62 ms on PlanetLab and 0.17 ms on the embedded environment. This means that the estimation on global reset time is around 102 ms for PlanetLab and 0.66 ms for the embedded environment.

6.2.4 Results

In this section we present the results we obtained from the evaluation. Figure 6.1 and Figure 6.2 show how the global reset time changes as the number of servers increases. We observe that the general trend is a linear increase in global reset time for both environments. The linear increase is expected for both environments since a linear increase in the number of servers means a linear increase in the CPU load due to message serialization and deserialization, which is the heaviest part of the global reset algorithm.

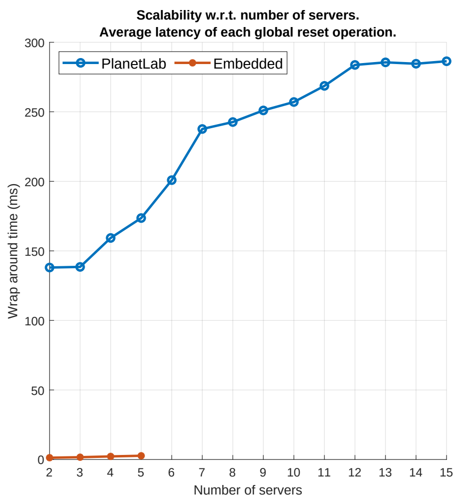

Figure 6.1: Scalability of global reset time with respect to the number of servers.

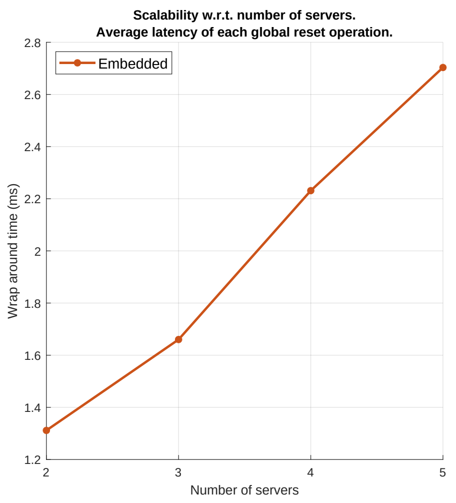

Figure 6.2: A zoomed version of Figure 6.1 that only shows the results for the embedded environment.

Although the general trend matches our expectations, there are some more surprising parts. In Figure 6.1, we see that in general, the latency grows faster when there are few servers compared to when there are many servers. We also see that the linear increase is much steeper for PlanetLab than for the embedded environment (in terms of absolute numbers). If the linear increase is majorly due to more message serialization and deserialization, the increase should be about the same for both environments.

In order to understand the two surprising phenomenon Figure 6.1 shows, we analyzed the 10 runs individually. One such run is shown in Figure 6.3. From the figure we can see that there are some big leaps from 3 to 4, from 6 to 7 and from 12 to 13. The reason for this is that due to the way the global reset algorithm is designed, all servers work in a lockstep manner and must wait for every other server before they can continue and change their phase. This means that the highest link latency among all the servers is the bottleneck of the global reset time. When it comes to Figure 6.3, the servers that are included into the system when the leaps happen have much higher link latency than the previously included servers. This is confirmed by analyzing multiple individual runs and we find that it is always the same slowest servers that cause the leaps. When this is clear, the “plateaus” following the leaps in Figure 6.3 are explained as follows. Since the slowest link is the bottleneck, after the inclusion of a server with a high link latency, the global reset time stays at the same level until a server with an even higher link latency is included.

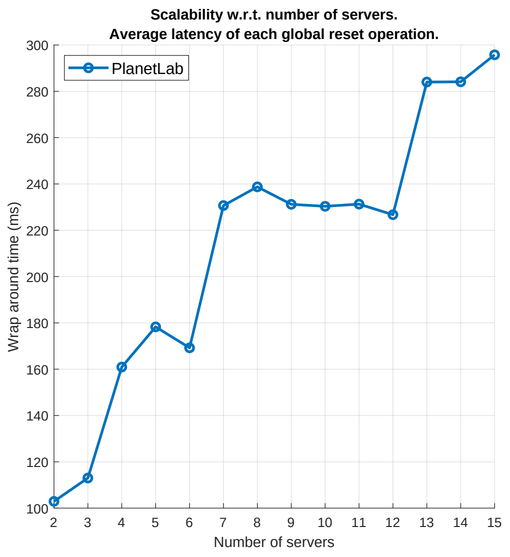

Figure 6.3: Scalability of Global reset time, but only the results from 1 of the 10 runs are shown.

With the insights from Figure 6.3, we can explain why the latency grows faster when there are few servers compared to when there are many servers. When there are only a few servers, the probability that the next server to be included has a higher link latency than all the previous servers is relatively large. But when many servers are already included, the probability that the next one has a longer link latency is instead relatively low. In other words, as we change from 4 to 5 servers, the 5:th server is relatively likely to have a slower link latency that the previous 4. But as we change from 11 to 12 servers, the 12:th server is not as likely to have a slower link latency than the previous 11. It is only when a server with a slower link latency is included that we see an increase in global reset time. Since the probability of including a slower server is larger when there are only a few severs, the global reset time growth is also larger when there are few servers compared to many servers. This reasoning is confirmed by Figure 6.1.

Using the insights from Figure 6.3, we can also understand why PlanetLab has a steeper increase than the embedded environment. Recall that as more servers are added, on average, the link latency of *the slowest* of them increases. And because it is the slowest server that determines the global reset time, the global reset time too increases for this reason as more servers are added. The link latency differences between the PlanetLab servers, which cause the global reset time increases, are *larger* than the extra overhead of having each server serialize and deserialize more messages. For the embedded environment, the increase is due to more serialization and deserialization, but for PlanetLab, the increase is also due to this described effect, which clearly has a bigger impact than more serialization and deserialization.

It is also worth noting that the results from neither of the two environments contradict the estimation in Section 6.2.3.

6.3 Conclusions

In this chapter, we presented the Global Reset technique. Global Reset can be used to bound unbounded counters, which is useful for many self-stabilizing algorithms. It can also be possible to bound the unbounded latency growth of [22] that was discovered in Chapter 4. Furthermore, we experimentally evaluated the time it takes for a global reset, and found that it grows linearly with the number of servers, although the global reset time is heavily affected by the slowest link speed in the system.

7 Conclusions

This thesis focused on the performance evaluation of [22]. Specifically, our evaluation criterion considered client latency. We based our study on an analytical estimation of the latency as well as a high-quality pilot implementation in the Rust programming language. Moreover, since [22] considers both synchronous and asynchronous system executions, we studied our pilot implementation in two different environments: PlanetLab and an embedded environment consisting of Raspberry Pis, where the former has long link latency and varying node speeds, while the latter has short link latency and identical node speeds.

Our results show that the client latency of [22] scales linearly with the number of completed requests, the number of servers and the number of clients. The performance behavior of [22] is similar in both environments, although the network latency plays a big role. Our analytical bound shows that for $n \geq 3$ servers, the lowest possible client latency is 5 message trips of the network. This is also confirmed by the experimental results.

In regard to the unbounded growth of client latency, as more and more requests are completed, as well as the unbounded counters and buffers of [22], we offered the use of a global reset algorithm [51]. In addition to a detailed evaluation of the algorithm, we explained how this self-stabilizing algorithm can fit in the analytical framework of [22]. The evaluation of the global reset algorithm as well as our analytical latency estimation of it show that the global reset latency is 4 times the slowest round trip time between the nodes in the system. As future work, we offer the reader to design new self-stabilizing algorithms that have unbounded counters and then transform them into ones that use bounded counters using our global system reset scheme (Chapter 6). Some examples of applying this technique can be found in [52, 62, 63] where we bound not only the counters but also the communication queue and the buffer size.

Bibliography

[1] ISO 26262-2018. Road vehicles — Functional safety. Standard, International Organization for Standardization, 2018.

[2] ISO 26262:6-2011. Road vehicles — Functional safety -— Part 6: Product development at the software level. Standard, International Organization for Standardization, 2011.

[3] ISO 26262:6-2018. Road vehicles — Functional safety -— Part 6: Product development at the software level. Standard, International Organization for Standardization, 2018.

[4] L. Alvisi and J. Martin. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(03):202–215, jul 2006.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[6] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Pearson, 2nd edition, 2005.

[7] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[8] Alexander Binun, Thierry Coupaye, Shlomi Dolev, Mohammed Kassi-Lahlou, Marc Lacoste, Alex Palesandro, Reuven Yagel, and Leonid Yankulin. Self-stabilizing byzantine-tolerant distributed replicated state machine. In Borzoo Bonakdarpour and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 36–53, Cham, 2016. Springer International Publishing.

[9] Olga Brukman, Shlomi Dolev, Yinnon A. Haviv, Limor Lahiani, Ronen I. Kat, Elad Michael Schiller, Nir Tzachar, and Reuven Yagel. Self-stabilization from theory to practice. *Bulletin of the EATCS*, 94:130–150, 2008.

[10] Viveck R. Cadambe, Nancy Lynch, Muriel Mèdard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. *Distrib. Comput.*, 30(1):49–73, February 2017.

[11] Antonio Casimiro, Jörg Kaiser, Johan Karlsson, Elad Michael Schiller, Philippas Tsigas, Pedro Costa, José Parizi, Rolf Johansson, and Renato Librino. Brief announcement: KARYON: towards safety kernels for cooperative vehicular systems. In Richa and Scheideler [70], pages 232–235.

[12] Antonio Casimiro, Jörg Kaiser, Elad Schiller, Pedro Costa, José Parizi, Rolf Johansson, and Renato Librino. The KARYON project: Predictable and safe coordination in cooperative vehicular systems. In *43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop, DSN Workshops 2013, Budapest, Hungary, June 24-27, 2013*, pages 1–12. IEEE Computer Society, 2013.

[13] Antonio Casimiro, Oscar Morales Ponce, Thomas Petig, and Elad Michael Schiller. Vehicular coordination via a safety kernel in the gulliver test-bed. In *34th International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops), Madrid, Spain, June 30 - July 3, 2014*, pages 167–176. IEEE Computer Society, 2014.

[14] Antonio Casimiro, José Rufino, Ricardo C. Pinto, Eric Vial, Elad Michael Schiller, Oscar Morales Ponce, and Thomas Petig. A kernel-based architecture for safe cooperative vehicular functions. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 228–237. IEEE, 2014.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst*, 20(4):398–461, 2002.

[16] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[17] Brent N. Chun, David E. Culler, Timothy Roscoe, Andy C. Bavier, Larry L. Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *Computer Communication Review*, 33(3):3–12, 2003.

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

[19] Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 29(9):2033–2045, 2018.

[20] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

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

[22] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Self-stabilizing byzantine tolerant replicated state machine based on failure detectors. In Itai Dinur, Shlomi Dolev, and Sachin Lodha, editors, *Cyber Security Cryptography and Machine Learning*, pages 84–100, Cham, 2018. Springer International Publishing.

[23] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Practically stabilizing virtual synchrony. *CoRR*, abs/1502.05183, 2015.

[24] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing virtual synchrony. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, volume 9212 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2015.

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

[26] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alexander A. Shvartsman, and Jennifer L. Welch. Brief announcement: virtual mobile nodes for mobile ad hoc networks. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, page 385. ACM, 2004.

[27] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alexander A. Shvartsman, and Jennifer L. Welch. Virtual mobile nodes for mobile ad hoc networks. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2004.

[28] Shlomi Dolev, Seth Gilbert, Elad Schiller, Alexander A. Shvartsman, and Jennifer L. Welch. Autonomous virtual mobile nodes. In Phillip B. Gibbons and Paul G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, page 215. ACM, 2005.

[29] Shlomi Dolev, Seth Gilbert, Elad Schiller, Alexander A. Shvartsman, and Jen-

nifer L. Welch. Autonomous virtual mobile nodes. In Suman Banerjee and Samrat Ganguly, editors, *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing, Cologne, Germany, September 2, 2005*, pages 62–69. ACM, 2005.

[30] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In Richa and Scheideler [70], pages 133–147.

[31] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing automatic repeat request algorithms for (bounded capacity, omitting, duplicating and non-fifo) computer networks. *CoRR*, abs/2001.3196015, 2020.

[32] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. In Alexander A. Shvartsman, editor, *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*, volume 4305 of *Lecture Notes in Computer Science*, pages 45–63. Springer, 2006.

[33] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8):884–900, 2010.

[34] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. Self-stabilizing byzantine resilient topology discovery and message delivery. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, volume 8255 of *Lecture Notes in Computer Science*, pages 351–353. Springer, 2013.

[35] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. Self-stabilizing byzantine resilient topology discovery and message delivery - (extended abstract). In Vincent Gramoli and Rachid Guerraoui, editors, *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, volume 7853 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2013.

[36] Shlomi Dolev, Panagiota N. Panagopoulou, Mikaël Rabie, Elad Michael Schiller, and Paul G. Spirakis. Rationality authority for provable rational behavior. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 289–290. ACM, 2011.

[37] Shlomi Dolev, Panagiota N. Panagopoulou, Mikaël Rabie, Elad Michael

Schiller, and Paul G. Spirakis. Rationality authority for provable rational behavior. In Christos D. Zaroliagis, Grammati E. Pantziou, and Spyros C. Kontogiannis, editors, *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, volume 9295 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2015.

[38] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. Brief announcement: Robust and private distributed shared atomic memory in message passing networks. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 311–313. ACM, 2015.

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

[40] Shlomi Dolev and Elad Schiller. Communication adaptive self-stabilizing group membership service. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):709–720, 2003.

[41] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Inf.*, 40(9):609–636, 2004.

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

[43] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Game authority for robust andscalable distributed selfish-computer systems. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 356–357. ACM, 2007.

[44] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Strategies for repeated games with subsystem takeovers: implementable by deterministic and self-stabilizing automata (extended abstract). In Antonio Manzalini, editor, *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems, Autonomics 2008, September 23-25, 2008, Turin, Italy*, page 37. ICST, 2008.

[45] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Game authority for robust and scalable distributed selfish-computer systems. *Theor. Comput. Sci.*, 411(26-28):2459–2466, 2010.

[46] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Robust and scalable middleware for selfish-computer systems. *Comput. Sci.*

Rev., 5(1):69–84, 2011.

[47] Shlomi Dolev, Elad Michael Schiller, Paul G. Spirakis, and Philippas Tsigas. Strategies for repeated games with subsystem takeovers implementable by deterministic and self-stabilising automata. *IJAACS*, 4(1):4–38, 2011.

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

[49] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In Jan Hlavička, Erik Maehle, and András Pataricza, editors, *Dependable Computing — EDCC-3*, pages 71–87, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[50] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[51] Chryssis Georgiou, Robert Gustafsson, Andreas Lindhe, and Elad Michael Schiller. Self-stabilization overhead: an experimental case study on coded atomic storage. *CoRR*, abs/1807.07901, 2018.

[52] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems*, pages 113–130, Cham, 2019. Springer International Publishing.

[53] Chryssis Georgiou and Nicolas C. Nicolaou. On the practicality of atomic MWMR register implementations. *CoRR*, abs/1111.2693, 2011.

[54] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[55] Jaap-Henk Hoepman, Andreas Larsson, Elad Michael Schiller, and Philippas Tsigas. Secure and self-stabilizing clock synchronization in sensor networks. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings*, volume 4838 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2007.

[56] Jaap-Henk Hoepman, Andreas Larsson, Elad Michael Schiller, and Philippas Tsigas. Secure and self-stabilizing clock synchronization in sensor networks. *Theor. Comput. Sci.*, 412(40):5631–5647, 2011.

[57] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX an-*

nual technical conference, volume 8, 2010.

[58] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* Online, 2018.

[59] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. volume 51, pages 45–58, 01 2007.

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

[61] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, page 10, USA, 2007. USENIX Association.

[62] Oskar Lundström, Michel Raynal, and Elad M. Schiller. Self-stabilizing set-constrained delivery broadcast. In *International Conference on Distributed Computing Systems*, 2020. to appear.

[63] Oskar Lundström, Michel Raynal, and Elad M. Schiller. Self-stabilizing uniform reliable broadcast. In *The International Conference on Networked Systems*, 2020. to appear.

[64] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 272–281. IEEE Computer Society, 1997.

[65] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[66] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report.

[67] Axel Nikalsson and Therese Petersson. Implementing Self-stabilizing Byzantine Fault-tolerant State-machine Replication: a Proof of Concept, Validation and Preliminary Evaluation. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, SE-412 96 Göteborg, Sweden, 2019.

[68] Axel Niklasson. plcli - a tool for running distributed applications on planetlab. In *2nd Workshop on Advanced tools, programming languages, and PLatforms for*

Implementing and Evaluating algorithms for Distributed systems, (ApPLIED) 2019, Proceedings, pages 66–74, 2019.

[69] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer, 2018.

[70] Andréa W. Richa and Christian Scheideler, editors. *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, volume 7596 of *Lecture Notes in Computer Science.* Springer, 2012.

[71] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review*, 35(5):15–28, 2001.

[72] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems.* 3rd edition, 2017.

A

Details of the Global Reset technique

In this appendix, we present the Global Reset technique in more detail. We do this by showing how to apply the Global Reset technique to MW-ABD [53], an algorithm for shared memory emulation. We show how to make MW-ABD self-stabilizing, including making it bounded.

We incrementally present our self-stabilizing bounded MW-ABD. We do this by first presenting the non-self-stabilizing unbounded MW-ABD in [53]. Then we show how to make that version self-stabilizing (but unbounded). Finally, we show our self-stabilizing bounded MW-ABD, which uses the global reset algorithm [51].

A.1 Non-self-stabilizing unbounded MW-ABD

MW-ABD is an algorithm for shared memory emulation, a concept already mentioned in Section 5.3.1. In some distributed systems such as the Internet, nodes communicate by sending messages to each other. However, some algorithms are easier to design for distributed systems if the nodes instead communicate by reading and writing to a shared memory location. Shared memory emulation is depicted in Figure 5.4.

A pseudocode description of MW-ABD appears in Algorithm 4, and is our interpretation of MW-ABD from [53]. Note that Algorithm 4 is not self-stabilizing and uses unbounded integers. The algorithm has a server-side and a client-side. Nodes that are designated as servers run the server-side while nodes that are designated as clients run the client-side.

The local variable of the server-side. At the heart of Algorithm 4's server-side is the local variable *pair* (line 62). *pair* is a 2-tuple containing *tag* and *value*. *value* represents this server's perception of what the current value of the emulated shared memory is, and *tag* is a timestamp that tells the server how recent this perception is. *tag* in turn is also a 2-tuple, containing *counter* and *id*, both which are integers. Tags are compared by lexicographical order, i.e. $tag1 > tag2 \iff tag1.counter >$

Algorithm 4: Our interpretation of the non-self-stabilizing unbounded MW-ABD algprithm of [53]. Code for node p_i.

```
61 Server-side:
62 local variable: $pair = \langle tag, value \rangle$, where $tag = \langle counter, id \rangle$;
63 upon QUERY arrival from $p_j$ begin
64     send QUERYack($pair$) to $p_j$;
65 upon WRITE($pairJ$) arrival from $p_j$ begin
66     if $pairJ.tag > pair.tag$ then $pair \leftarrow pairJ$;
67     send WRITEack($pair$) to $p_j$;
68 upon INFORM($pairJ$) arrival from $p_j$ begin
69     if $pairJ.tag > pair.tag$ then $pair \leftarrow pairJ$;
70     send INFORMack($pair$) to $p_j$;
71
72 Client-side:
73 function read() begin
74     send QUERY to all servers and wait for a majority of replies;
75     let $maxPair$ be the $pair$ with the largest $pair.tag$ received in the previous line;
76     send INFORM($maxPair$) to all servers and wait for a majority of replies;
77     return $maxPair.value$;
78
79 function write(v) begin
80     send QUERY to all servers and wait for a majority of replies;
81     let $maxPair$ be the $pair$ with the largest $pair.tag$ received in the previous line;
82     let $newPair := \langle \langle maxPair.tag.counter + 1, i \rangle, v \rangle$;
83     send WRITE($newPair$) to all servers and wait for a majority of replies;
```

$tag2.counter \lor (tag1.counter = tag2.counter \land tag1.id > tag2.id)$. (The reason for having *id* and not just *counter* will become apparent once the client-side has been presented.) In other words, the central idea of Algorithm 4 is that all servers store their perception of the current value, and how recent it is.

The client-side. Before continuing with the server-side, let us turn to the client-side, found in lines 72-83. A *read*() operation (line 73) consists of first letting the client send a QUERY message to all servers and then waiting for a response from a majority of them (line 74). The purpose of doing this is for the client to discover the current value (in the emulated memory). Of the received pairs, the client finds the pair with the largest tag (line 75) and then sends an INFORM message to all servers, and again waits for a majority of them to reply (line 76). The purpose of this inform phase is to guarantee linearizability [54] of reads and writes, of which more details can be found in [5, 53]. After the inform phase, the value with the maximum tag found during the query phase is returned (line 77). Why majorities are used will be presented after we have presented the *write*() operation.

The *write*() operation (line 79) works in a similar way as the *read*() operation. The client first sends a QUERY message to all servers and waits for a majority to reply (line 80). The purpose of this is for the client to discover the maximum tag currently in the system (line 81). Then the client creates a new pair, consisting of the maximum tag's counter, the client's own id and the value the client wants to write (line 82). By adding one to the current maximum tag's counter, the client is sure that this new tag is larger than all other tags, and hence the value that the client is currently writing will be considered to be the most recent value written (until a new write happens). Finally, the client sends a WRITE message to all servers and waits for a majority to respond (line 83).

At this point, we can explain why the tags contain *id* and not only *counter*. If two clients perform *write*() operations concurrently, it might happen that after their query is done (they are at line 81), they calculate the same *maxPair*. If *id*s are not used to distinguish between these two client writes, two values will have *the same tag*. This means that some servers might store one value and some servers the other, causing inconsistency. But with *id* as a tie-breaker, this inconsistency is avoided.

As for why majorities are used, the reason is to be able to be as fault tolerant as possible while still guaranteeing linearizability [54]. If a client is not required to wait for replies from *all* servers, but only from a *majority*, it means that a subset of the servers are allowed to crash while still allowing the clients to be served. However, if only required to wait for less than a majority, there can be inconsistencies, including violation of linearizability. For example, suppose the clients only need to wait for replies from a quarter of the servers. Then client c_1 can query and write to one quarter, while client c_2 can query and read from another quarter, with no intersection between them. This means that c_2 can miss out on writes that c_1 has performed, which is unwanted inconsistency. With majorities, the above cannot happen, since all majorities of servers intersect. This means that there would be at least one server that c_1 writes to that also c_2 reads from, making the above inconsistency impossible.

The server-side. The server-side of Algorithm 4 appears in lines 61-70. When a WRITE or INFORM message arrives (lines 65 and 68), the server simply updates its locally stored *pair* if the received *pairJ* is greater than *pair* (line 66 and 69). For all three types of messages, the server the responds with an acknowledgement containing its locally stored *pair* (lines 64, 67 and 70).

A.2 Self-stabilizing unbounded MW-ABD

Algorithm 5 is a self-stabilizing (but still unbounded) version of Algorithm 4 that has been created by us. The boxed lines mark the changes compared to Algorithm 4. The major difference between Algorithm 4 and 5 is that the latter includes gossiping and some changes to the query mechanism, which are needed for recovery from arbitrary transient faults. Let us now describe these changes in more detail.

Gossiped values are stored in the new local array variable *gossip* (line 86), and gossip messages are sent and received in line 99 and 97, respectively. The array

Algorithm 5: Self-stabilizing unbounded MW-ABD. Code for node p_i. The boxed lines mark code added compared to Algorithm 4.

```
84  Server-side:
85  local variables: $pair = \langle tag, value \rangle$, where $tag = \langle counter, id \rangle$;
86  $\boxed{gossip \text{ array with one entry for each server;}}$
87  upon QUERY($\boxed{rw}$) arrival from $p_j$ begin
88      $\boxed{\textbf{if } rw = \text{'r' } \textbf{then}}$ send QUERYack($pair$) to $p_j$;
89      $\boxed{\textbf{if } rw = \text{'w' } \textbf{then send } \text{QUERYack}(max(\{pair.tag\} \cup \bigcup_{k \in \text{all servers}} gossip[k])) \text{ to } p_j;}$

90  upon WRITE($pairJ$) arrival from $p_j$ begin
91      if $pairJ.tag > pair.tag$ then $pair \leftarrow pairJ$;
92      send WRITEack($pair$) to $p_j$;
93  upon INFORM($pairJ$) arrival from $p_j$ begin
94      if $pairJ.tag > pair.tag$ then $pair \leftarrow p$;
95      send INFORMack($pairJ$) to $p_j$;
96  upon GOSSIP($tagJ, tagI$) arrival from $p_j$ begin
97      $\boxed{(gossip[i],\ gossip[j]) \leftarrow (max\{gossip[i], pair.seq, tagI\},\ max\{gossip[j], tagJ\});}$
98  do forever begin
99      $\boxed{\textbf{foreach} \text{ server } p_j \textbf{ do send } \text{GOSSIP}(max\{gossip[i], pair.seq\},\ gossip[j]) \text{ to } p_j;}$

100
101 Client-side:
102 function read() begin
103     send QUERY($\boxed{\text{'r'}}$) to all servers and wait for a majority of replies;
104     let $maxPair$ be the $pair$ with the largest $pair.tag$ received in the previous line;
105     send INFORM($maxPair$) to all servers and wait for a majority of replies;
106     return $maxPair.value$;

107
108 function write(v) begin
109     send QUERY($\boxed{\text{'w'}}$) to all servers and wait for a majority of replies;
110     let $maxPair$ be the $pair$ with the largest $pair.tag$ received in the previous line;
111     let $newPair := \langle \langle maxPair.tag.counter + 1, i \rangle, v \rangle$;
112     send WRITE($newPair$) to all servers and wait for a majority of replies;
```

gossip has an entry for each other server, where p_i stores the tag the respective server has gossiped. Gossip messages are sent in line 99, where p_i sends to p_j the maximum of its own gossip entry, $gossip[i]$, and $pair.tag$, as well as the tag it stores about p_j. The former is for p_i to inform other servers what p_i stores in $pair.tag$, and the latter is for p_j to make sure that no other server stores a larger value about p_j than p_j itself. This becomes evident in line 97, where gossip messages are received. There, p_i ensures that $gossip[i]$ is at least as large as what p_j stores about it. It also updates $gossip[j]$ if the received value is larger.

Gossiping solves the following arbitrary transient fault: a single server p_k gets its *pair* corrupted so that it contains a very large *tag*. When clients read and write, it might happen that they only communicate with majorities excluding p_k. But if a read suddenly gets a reply from p_k, the tag returned is greater than all other servers' tags, and the read returns the value p_k stores. This value is not necessarily the same as what was most recently written, which is a violation of linearizability, which MW-ABD should provide. But with gossiping, every server stores p_k's tag in their *gossip* array. This means that in line 89, even the large tag of p_k is returned, and in line 111, the client makes sure its write gets a tag larger than p_k's. Thus, the previously outlined scenario can no longer happen, and linearizability is preserved even after arbitrary transient faults have occurred.

A.3 Self-stabilizing bounded MW-ABD using Global Reset

Algorithm 6 is a bounded version of Algorithm 5, and uses the global reset algorithm by Georgiou *et al.* [51]. The boxed lines mark code added compared to Algorithm 5. Our modifications follow how [39] created a bounded variation.

The first difference between Algorithm 6 and Algorithm 5 concerns how the server-side handles incoming messages when *pair.tag.counter* has reached MAXINT. We see that in lines 117, 121 and 125, the server replies with a BUSY message instead of an acknowledgement if MAXINT has been reached. This is because if MAXINT has been reached, the server must reset *counter* to 0 before new client requests can be served. We also see that if a client receives a BUSY message, the client sleeps `backOffTime` and then restarts the *read*() or *write*() operation from the beginning (lines 138, 140, 144 and 147). The operation must start from the beginning since *maxPair* (line 139) or *maxTag* (line 145) might contain a value close to MAXINT. Also, if between lines 138 and 140 (or 144 and 147), the servers reset *counter* to 0, a value close to MAXINT is immediately sent by the client again in lines 140 or 147, which means that a second reset happens in very close succession. To prevent this, clients sleep `backOffTime`, which is an upper bound on the time it takes for the servers to reset.

The second difference between Algorithm 6 and Algorithm 5 concerns lines 132 and 134. The former line checks that if all servers store MAXINT, this server calls *propose*(*pair.value*) in order to initiate a global reset, proposing to store its own value after the reset. The latter line handles the call to *localReset*(*value*) that [51] makes. The line sets all entries in *gossip* to the smallest possible tag and sets the server's own *pair* to the value *localReset*() was called with and the smallest possible tag.

Algorithm 6: Self-stabilizing bounded MW-ABD. Code for node p_i. The boxed lines mark code added compared to Algorithm 5.

```
113 Server-side:
114 local variables: $pair = \langle tag, value\rangle$, where $tag = \langle counter, id\rangle$;
115 $gossip$ array with one entry for each server;
116 upon QUERY($rw$) arrival from $p_j$ begin
117     if $\boxed{pair.tag.counter \geq \text{MAXINT}}$ then $\boxed{\textbf{send}\ \text{BUSY to}\ p_j}$ else
118         if $rw$ = 'r' then send QUERYack($pair$) to $p_j$;
119         if $rw$ = 'w' then send QUERYack($max(\{pair.tag\} \cup \bigcup_{k \in \text{all servers}} gossip[k])$)
            to $p_j$;
120 upon WRITE($pairJ$) arrival from $p_j$ begin
121     if $\boxed{pair.tag.counter \geq \text{MAXINT}}$ then $\boxed{\textbf{send}\ \text{BUSY to}\ p_j}$ else
122         if $pairJ.tag > pair.tag$ then $pair \leftarrow pairJ$;
123         send WRITEack($pair$) to $p_j$;
124 upon INFORM($pairJ$) arrival from $p_j$ begin
125     if $\boxed{pair.tag.counter \geq \text{MAXINT}}$ then $\boxed{\textbf{send}\ \text{BUSY to}\ p_j;}$ else
126         if $pairJ.tag > pair.tag$ then $pair \leftarrow pairJ$;
127         send INFORMack($pair$) to $p_j$;
128 upon GOSSIP($tagJ, tagI$) arrival from $p_j$ begin
129     $(gossip[i],\ gossip[j]) \leftarrow (max\{gossip[i], pair.seq, tagI\},\ max\{gossip[j], tagJ\})$;
130 do forever begin
131     foreach server $p_j$ do send GOSSIP($max\{gossip[i], pair.seq\}$, $gossip[j]$) to $p_j$;
132     $\boxed{\textbf{if}\ \forall p_k : gossip[k] \geq \text{MAXINT}\ \textbf{then}\ [51].propose(pair.value);}$
133 upon localReset($value$) called by [51] begin
134     $\boxed{\textbf{foreach}\ \text{server}\ p_j\ \textbf{do}\ gossip[j] \leftarrow \langle 0,0\rangle;\ pair \leftarrow \langle\langle 0,0\rangle, value\rangle;}$
135
136 Client-side:
137 function read() begin
138     send QUERY('r') to all servers and wait for a majority of replies,
          $\boxed{\text{or if BUSY received, sleep backOffTime, then restart from line 138}}$;
139     let $maxPair$ be the $pair$ with the largest $pair.tag$ received in the previous line;
140     send INFORM($maxPair$) to all servers and wait for a majority of replies,
          $\boxed{\text{or if BUSY received, sleep backOffTime, then restart from line 138}}$;
141     return $maxPair.value$;
142
143 function write(v) begin
144     send QUERY('w') to all servers and wait for a majority of replies,
          $\boxed{\text{or if BUSY received, sleep backOffTime, then restart from line 144}}$;
145     let $\boxed{maxTag\ \text{be the largest}\ tag}$ received in the previous line;
146     let $newPair := \langle\langle \boxed{\text{maxTag}}.counter + 1, i\rangle, v\rangle$;
147     send WRITE($newPair$) to all servers and wait for a majority of replies,
          $\boxed{\text{or if BUSY received, sleep backOffTime, then restart from line 144}}$;
```