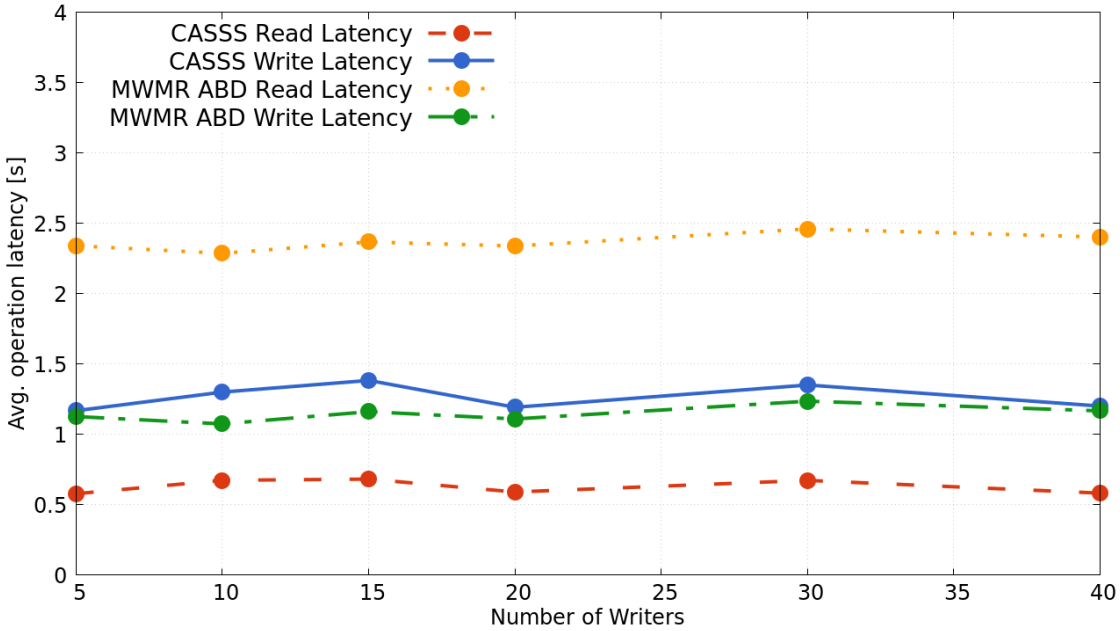
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Self-Stabilizing Services for Emulating Distributed Shared Memory on Message Passing Platforms

Master's thesis in Computer Systems and Networks

ROBERT GUSTAFSSON

ANDREAS LINDHÉ

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Self-Stabilizing Services for Emulating Distributed Shared Memory on Message Passing Platforms

ROBERT GUSTAFSSON
ANDREAS LINDHÉ

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Self-Stabilizing Services for Emulating Distributed Shared Memory on Message Passing Platforms
ROBERT GUSTAFSSON
ANDREAS LINDHÉ

Supervisor: Elad M. Schiller, Dept. of Computer Science and Engineering
Examiner: Olaf Landsiedel, Dept. of Computer Science and Engineering
External Examiner: Chryssis Georgiou, Dept. of Computer Science, University of Cyprus

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Operation latency with respect to the number of concurrent writers. The number of readers and servers are fixed to ten. The number of writers ranges from five to 40.

Typeset in LaTeX
Gothenburg, Sweden 2018

Self-Stabilizing Services for Emulating Distributed Shared Memory on Message Passing Platforms

ROBERT GUSTAFSSON
ANDREAS LINDHÉ
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We use self-stabilization techniques to construct a distributed service for shared memory emulation on message passing platforms. We are the first to implement and practically evaluate the self-stabilizing algorithm for atomic shared memory emulation developed by Dolev, Petig and Schiller, which in turn is the first algorithm of its kind to address privacy, malicious behaviour and self-stabilization. Furthermore, we have used techniques from the Self-Stabilizing Reconfiguration paper by Dolev et al. to create a mechanism for performing a virtually synchronous global reset of the entire system to deal with transient faults.

With a firm analytical basis, these algorithms provide the tools needed to deal with arbitrary starting configurations and recover to legal behaviour within a bounded time. To show the applicability and correctness in practice, we have created an evaluation environment using PlanetLab. The evaluation shows that our implementation of the self-stabilizing version of the Coded Shared Atomic Memory algorithm (CAS) scales very well both in terms of the number of servers and in terms of the number of concurrent clients. It is shown to have only a constant overhead compared to the traditional CAS algorithm. Furthermore, the evaluation shows that it scales well with respect to data object size too—the system shows almost no slowdown for data objects up to 512 KiB, and is only slightly slower for data objects up to 1 MiB. Last but not least, the evaluation reveals that the global reset mechanism, which is the worst case scenario for handling transient faults, is as fast as a few client operations. For systems with up to 20 servers, the global reset is done within a few seconds.

The same techniques used in this project can also be used to create a multitude of other self-stabilizing services and algorithms, such as self-stabilizing Paxos and self-stabilizing Virtual Synchrony to name a few.

Keywords: distributed systems, distributed shared memory emulation, self-stabilizing algorithms

Acknowledgements

We want to thank our supervisor Elad Michael Schiller for providing his expertise and working closely with us throughout the project.

We also want to thank Chryssis Georgiou for providing guidance in constructing the evaluation experiments, as well as assisting in our analysis of the results.

Lastly, we want to thank Combitech, for very generously providing us with a great working environment in their office at Lindholmen.

Contents

List of Algorithms

List of Figures

List of Tables

1

Introduction

The concept of shared memory emulation is a cornerstone in distributed computing. Many problems—e.g., consensus—can be more easily resolved in the shared memory paradigm than using message passing. While any shared memory solution can be ported into a solution using message passing, many algorithms are designed to use shared memory because of the performance and ease-of-use that it brings on non-distributed systems. For some applications, switching from shared memory to message passing might entail unwanted overhead and large development costs. To abstract away the complex task of developing a new, application specific, message passing solution, an alternative is to emulate shared memory on a message passing platform. A library can handle the message passing and provide the application with `read()` and `write(x)` function calls, making it seem as if the system was operating on a shared memory.

Shared memory emulation can either be used as a fault-tolerant and highly available distributed storage solution or as a low-level synchronization primitive. No matter the context, it is desirable to have a solution with low communication cost and storage cost which can guarantee apparent atomicity of the operations. Prominent examples of where such techniques can be used are cloud computing and cloud storage. There are also new emerging markets for such technologies, such as the area of autonomous vehicles, which produces large amounts of data during operation [2]. In order to process such large amounts of data, fog or cloud solutions might be the only viable solution. But of course, that would require a high performance, reliable solution to store all that data, in the asynchronous context of highly mobile vehicular systems.

Distributed systems have many advantages over centralized solutions such as data redundancy and greater availability. While many systems can handle server crashes by redundancy, there are rare faults that can cause the system to malfunction inde-

terminately. Reaching such a state may require external (human) intervention for triggering a complete restart of the system, which might not be feasible in a large, distributed system. Self-stabilization is a powerful technique for fault tolerance, which provides guarantees that the system will always return to a well behaved state within a bounded time period. In this project, we study and implement self-stabilizing solutions for shared memory emulation on asynchronous message passing systems.

1.1 Background

Sharing a data object among decentralized servers to provide the functionality of a distributed storage has been a topic within research for decades. The problem that has been studied, is to emulate a shared memory that has the property that operations are by all appearances atomic (linearisable), either in a single-writer multi-reader (SWMR) or multi-writer multi-reader (MWMR) setting. One of the earliest and most significant contributions is [3] by Attiya, Bar-Noy and Dolev, later known as the ABD algorithm, which uses a tagged data objects in a quorum system with full replication, to achieve fault-tolerance with high availability while still being able to guarantee atomicity. It has since been iterated upon by Lynch and Shvartsman [4] (amongst others), making it work in a MWMR setting.

For small data objects, algorithms like these are quite efficient but do not scale well when it comes to larger data objects. This is because of the full replication to all servers in the system. In 2017, Cadambe et al. proposed a solution to this, through the *Coded Atomic Storage* (CAS) algorithm [5]. The CAS algorithm uses erasure coding in order to achieve data redundancy but with much lower communication cost compared with algorithms that use full replication.

Although CAS provides an efficient solution to the problem of distributed data replication, and provide tolerance against crash prone servers, it may end up in a situation it can not recover from. While server crashes and lost messages are in the fault model of the system, there is a type of fault called *transient fault*. Such a fault is something that happens incredibly rarely, but once it happens it can cause the system configuration to enter any arbitrary state. Most algorithms do not account for such faults, since they are rare and are hard to deal with. This is where another concept of fault-tolerance comes in, which is called *self-stabilization*. A self-stabilizing system will always converge to a safe state within a bounded period of time, with no intervention needed. So while the transient fault

will cause a recovery period where the system might be broken, it is guaranteed to eventually return to normal operation (called a *legal execution*; LE).

In [6], Dolev, Petig and Schiller present a self-stabilizing version of CAS, which provides fault-tolerance against malicious behaviour, privacy and self-stabilization. In this report, we call their algorithm *CASSS*, for Coded Atomic Storage Self-Stabilization. The authors provides theoretical proofs for the correctness and complexity bounds of the algorithm, and we are the first to practically implement and verify their algorithm for self-stabilizing CAS. We do this in the asynchronous setting of message passing systems on the Internet, where messages may be lost in transit, reordered or even maliciously corrupted.

1.2 Related Work

We know of no published work which implements and evaluates self-stabilizing MWMR algorithms for emulation of shared memory. But there are many interesting works which relates to this project, and in this section we take a look at a number of those works.

Dolev et al. proposes a pseudo self-stabilizing version of SWMR ABD in [7]. The authors extends the ABD algorithm to tolerate *transient and permanent faults*, but with a weaker notion of self-stabilization than [6] uses. A pseudo self-stabilizing algorithm guarantees convergence, but unlike proper self-stabilization it gives no bound on how long it will take to converge.

Vacana [8] at the University of Cyprus did a master thesis about implementation and evaluation of a self-stabilizing atomic read/write register service. The algorithms implemented were a non-self-stabilizing and a self-stabilizing version of SWMR ABD based on the work by Alon et al. [9].

Nicolaou and Georgiou [10] did an experimental evaluation of four MWMR register emulation algorithms. The algoritms evaluated were *SWF, APRX-SWF, CwFr* and *SIMPLE*. The SIMPLE algoritm is a MWMR version of ABD for generalized quorum systems and we use an alternative implementation with a self-stabilizing quorum system in this project.

Layered Data Replication, or LDR, is an algorithm by Fan and Lynch [11] for distributed storage which is optimized for large data objects. More specifically, it

is optimized for when the size of the data objects are considerably greater than the size of its metadata. The key idea is to separate the functionality among the servers in a layered structure. Some of the servers are so called *directory servers*, which only stores metadata about the objects. The actual objects are stored in the so called *replica servers*. To know which replica server a specific data object is stored on, a client first needs to query the directory servers and get a quorum of responses. This reduces the communication cost significantly compared to traditional replication based schemes like ABD, since read and write operations can be done in near constant time. Even though this gives algorithm gives particularly fast read operations, CAS is more flexible when it comes to cost efficiency versus redundancy.

The Layered Data Storage (LDS) algorithm by Konwar et al. is another algorithm which optimizes storage of large data objects by a dual layer approach. Unlike LDR, the clients in LDS never interact with the second layer; clients only interact with the first layer. Instead LDS adopts an approach inspired by the edge computing paradigm, by having the first layer of servers act as proxy servers which will cache objects from the second layer to be served down to the clients. It also adopts regenerative erasure coding to improve storage and communication costs.

Cadambe et al. [13] have recently published an algorithm called ARES (Adaptive, Reconfigurable, Erasure Coded, Atomic Storage), which supports reconfiguration of a shared memory emulation service that is based on erasure coding. The authors also presents the first atomic memory service that uses erasure coding with only two-rounds for a client operation. While combining these two create an efficient solution with liveness even during quorum collapses, it does not consider self-stabilization. Lynch and Shvartsman [14] presents another algorithm that also support reconfiguration, called RAMBO. The algorithm uses full replication and changing configuration requires that old members send the data to the new members.

RADON, standing for Repairable Atomic Data Objects in Networks, is an algorithm for distributed atomic storage by Konwar et al. [15]. Its read and write operations are nearly identical to those of ABD, but with a larger quorum size for write operations. RADON employs a self-repair mechanism to rebuild the lost data via erasure coding, making it resilient against server crashes while still providing atomicity and liveness. This should mean that it is particularly suitable in systems with high churn rate of servers. It does however not facilitate for multiple concurrent writers (because of its similarity to ABD), and is not self-stabilizing.

1.3 Our Contribution

We are the first to implement and practically evaluate the self-stabilizing algorithm for atomic shared memory emulation developed by Dolev, Petig and Schiller [6]. This is the first algorithm of its kind to address privacy, malicious behaviour and self-stabilization. We have also created a reset mechanism, based on principles from [1] by Dolev et al. The reset mechanism can perform a virtually synchronous global reset of the entire system. Additionally, we created a self-stabilizing reincarnation number service, which provides recyclable client identifiers.

To test the system during development, we created a virtualised platform based on Docker and NS-3, which provides a more realistic setting (with real communication delays and bandwidth limitations than just running it locally). A similar endeavour was made by Casparsson and Gardtman [16]. In order to show the applicability and correctness in practice, we have created an evaluation environment using PlanetLab. Our experiments on PlanetLab shows that the algorithms are indeed efficient enough to be used in practice, as demonstrated by our prototype.

The evaluation shows that our implementation of the self-stabilizing version of CAS scales very well when increasing the number of servers and clients respectively. The overhead for self-stabilization, in our experiments, is constant when compared to two implementations of the original CAS algorithm. The system shows almost no slowdown for data objects up to 512 KiB, and is only slightly slower for data objects up to 1 MiB. Last but not least, the evaluation reveals that the reset mechanism is almost as fast as a few client operations.

There are many other services and algorithms that make use of quorum systems and gossip services or has bounded tag numbers. We believe that the building blocks in this project could therefore be used to create other self-stabilizing services and algorithms.

1.3.1 Document Structure

The rest of this report is structured as follows. In Chapter 2, we introduce the system, models and evaluation criteria which are used. We then go into the theoretical background in Chapter 3, which is the existing foundation on which this project has been built. That includes the concept of quorum systems, what shared memory emulation is, the communication channel used as well as an outline of the

algorithms for shared memory emulation which are used in this project. After that, in Chapter 4, the newly developed algorithms for reincarnation and global reset are described. Chapter 5 and Chapter 6 go over how the system was implemented and how it was evaluated, respectively. The evaluation results are presented and discussed in Chapter 7, and Chapter 8 concludes the report by a short discussion regarding similar works, possible extensions as well as a conclusion.

2

System

In this chapter, the system setting and system models are described. In Section 2.1, the architecture and functionality of the system are explained. Some note worthy implementation decisions are discussed, and a description of the building blocks which make up the system. Section 2.2 outlines the definitions and assumptions regarding how communication, execution and faults are modelled.

2.1 System Setting

The system setting considers a network with N nodes which are referred to as servers and another set of nodes which are client nodes. Each of the servers has access to a register where records are stored. A record can hold either a coded element or an empty element (i.e., a `None` object). The original data object can be reassembled from any k, non-empty, elements.

Clients are nodes which interact with the shared-memory service using read and write operations. These operations include multiple communication rounds of requests and responses between a client and the servers. Every client performs operations sequentially, but operations can still be concurrent since clients acts independently of each other.

The basic architecture of the system is based on a client-server scheme, as seen in Figure 2.1. The servers are part of a fully connected network, since we are using transport layer protocols to communicate. Logical links are established using acknowledged, directed channels. Every server communicates with every other server using two such directed channels—one in each direction. The servers can at any time send or receive gossip messages over these channels, informing each other

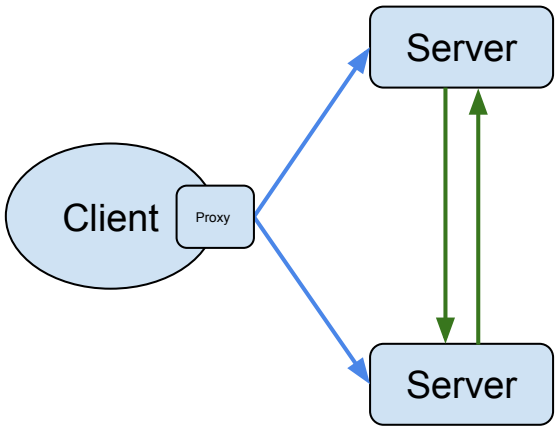


Figure 2.1: Diagram of the basic client-server architecture for the system. Arrows represent directed, acknowledged communication channels. The servers are in a fully connected cluster, and every client is connected to every server via a directed channel.

about the state of the data object or other meta data and configuration changes. The underlying communication on which the channels are implemented is assumed to be unreliable, meaning that packets may be omitted, reordered, duplicated or corrupted, and the channels must be able to deal with that.

Every client is connected to every server via a directed link. We call these links directed, even though they technically send information in both directions. While a server can piggyback a payload on the acknowledgement message to the client, the delivery of that payload is in turn not acknowledged. Therefore, we find it more useful to view it as a directional rather than bidirectional link.

The environment is asynchronous, and servers and clients may at any time fail-stop. In the absence of transient faults, servers can resume their operation at any moment, but clients can not. Instead, clients have to reincarnate, meaning that if they come back, they must do so under a new unique identifier (effectively becoming a new client).

2.1.1 Functionality

The service is implemented as a library, which can be used by applications in order to provide access the read and write operations. Calls to the functions `read()` and `write(x)` behaves just as if the service was an actual shared memory. Calls to these functions blocks the calling process until it returns. A successful read operation returns the data object, and a write operation blocks until it is done writing the object (and returns nothing).

Each node is initialized with a configuration file, which specifies the information needed for the system to work. The configuration file holds a list of participating servers, coding parameters, how many server failures can be tolerated, and the storage bounds.

2.1.2 Shared Objects

The functionality for the whole system is to emulate a shared memory. This memory contains a single data object and there are multiple readers and writers concurrently accessing the object. An important property that has to be satisfied for emulating a shared memory is atomicity. In other words, if a client reads two times in a row, the second read operation should never return an older value than the first read operation did. The key take-away is that concurrent operations should appear as if they were sequential (i.e., they are linearisable).

Unique tags are used to determine the causal relationship between writes, and are used to retrieve the most up-to-date version of the shared object. A tag is defined as a tuple of a sequence number and a writer's identification, which in turn consist of an incarnation number and a unique hardware address. The sequence number is used as an overall causal relationship identifier but concurrent writes with the same sequence number are deterministically sorted on the writer identification.

A record in a server's register is a tuple of the form $(tag, element, phase)$. Along with the unique tag and the coded element (see Section 3.5.1), there is a field called the Record's *phase* (which can be *pre*, *fin* or *FIN*). The record phases are discussed in more detail in Section 3.5.

2.1.3 Building Blocks

There are algorithms which has critical roles in the main algorithm, but are not themselves part of the main algorithm. We call these algorithms *building blocks*, and there are many alternatives for which building blocks to use when building the main algorithm. This section goes through what building blocks we have chosen and why. The building blocks are summarized below, and are covered more in-depth in Chapter 3 and Chapter 4.

A self-stabilizing system can not use a regular TCP connection to communicate,

since TCP is not self-stabilizing. Therefore, another protocol is required, which has both reliable and self-stabilizing end-to-end communication. The protocol we use is a self-stabilizing version of the token passing algorithm by Dolev in [17, Figure 4.1]. In our implementation of this token passing channel, both TCP and UDP can used for the data transfer (see Section 5.2.1).

A self-stabilizing gossip protocol is used between servers to periodically share the largest tag number for each of the phases: *pre, fin* and *FIN*. Gossip messages are delivered unreliably, and that is a feature. New gossip messages should overwrite old ones, if one message has not yet been delivered before the next one arrives.

Quorum systems are used to have good availability and to guarantee atomicity when servers can fail during operation. A quorum system rely on that enough information can be recovered from the intersection between subsets of servers. An external directory service which can serve the list of storage servers is assumed to exist. In our implementation, a configuration file is used. For more information on quorum systems, see Section 3.1.

A reset mechanism is needed in self-stabilizing distributed services, whenever there is any kind of sequence number present. Even if the sequence number counter in a message were to be chosen to be so large that it could never reasonably be expected to overflow (like a 64-bit integer), a transient fault could still cause the counter to jump to a number so large that it will overflow. To handle such a scenario, there is a wrap-around mechanism that should be triggered whenever MAXINT (e.g., $2^{64} - 1$) is observed.

An assumption in Dolev et al. [6] is that if clients fail they stop and never return to operation. But [6] mentions as a possible extension that recyclable client identifiers could be used to overcome that problem. If a client crashes during an operation and then comes back again, it will be treated as if it was a completely new client joining. When a client rejoin, it must have a new globally unique identification number. To achieve that, a monotonically incrementing counter (called incarnation number) is appended to the client's unique identifier.

2.2 Models

In this section, we present the models that the system is based on. The models describe certain core concepts and approaches which are fundamental for under-

standing how and why the system works. These models define how the *communication*, *execution*, and *faults* are represented in the design of the system. Last but not least we give our definition of *self-stabilization*.

2.2.1 Communication Model

In this project, we use transport layer protocols for message passing. That means that routing does not concern us, and the network can be seen as a fully connected virtual topology. The model we use assumes an *asynchronous* setting where messages may be arbitrarily reordered, created or deleted during transit. Furthermore, we assume *communication fairness*, under the definition that if a node is trying to send a message over a link infinitely often it will eventually deliver the message. Simply put, this means that a link might be down for an arbitrary amount of time, but not forever. If the system were to be truly self-stabilizing, the underlying communication protocols would also need to be self-stabilizing.

2.2.2 Execution Model

A running instance of the program is said to be a *processor*, and can live either on the same machine as other processors or on a machine which is physically separated from others. A processor can take (atomic) steps according to the program code, and may only end up in an illegal state as a consequence of a transient fault. An *illegal state* is defined as any state such that the invariants of the system does not hold. A *state* is a configuration of all variables and any in-transit messages, at some given point in time. Transitions between states represent a step being taken, which can be either the departure or arrival of a message or an execution which changes one or more variables.

The execution model builds on a discrete time model which considers an asynchronous setting, where any step can be interleaved in any order with other steps (as longs as they are not causally related). Any step which can be taken at some point in time is guaranteed to eventually be taken, but no guarantee is given as to when that happens. This is called *fair execution*.

An *execution* is a sequence of steps. The set of steps which performs the desired task of a system is called the *legal execution* (LE). During the legal execution, no step can be taken which violates the system requirements.

2.2.3 Fault Model

Communication Faults are faults which may occur to messages during transit, as well as when sending or receiving a message. There are three types of communication faults which can occur: *omission*, *duplication*, and *reordering*. Omission means that a message was lost, duplication means that a received message may be received again, and reordering means that two messages may be in another order than they were sent in.

Node Faults may occur, in the form of a crash failure. Servers may crash and return to operation again at any moment, except that all servers are required to be alive at the time of a global reset. At most f servers are allowed to fail. Clients are allowed to crash and resurrect, but if they do they must use a new client identifier (see Section 4.1). Note that there is no way to externally distinguish between a crashed node and a node that has lost connectivity.

Transient Faults are assumed to occur only very rarely, but when they do they may cause the program to end up in an arbitrary state. Any violation of the system assumptions is considered a transient fault. Transient faults can for example be a soft error (such as a bit flip, perhaps induced by background radiation) or the very unlikely event of a CRC code failing to detect a bit error in a transmitted packet. What causes the error and what effect the error will have on the system is impossible to say, why it is impossible to protect against within traditional fault models.

2.2.4 Self-Stabilization

The concept of self-stabilization provides a strong fault-tolerance guarantee in that it will always recover from a transient fault. While there is no way of avoiding transient faults from occurring, a self-stabilizing system will return to correct behaviour within a bounded period of time (assuming the program code itself stays intact).

The self-stabilization model we use assumes that a processor may start in an arbitrary state caused by a transient fault, after which no more transient faults occur. Since the transient fault may cause the system to enter any conceivable

state without a good reason, the history leading up to that point is of no interest. Therefore, we do not consider any progress before the last instance of a transient fault. Every possible execution of a self-stabilizing algorithm should eventually lead to a set of steps which belongs to LE. LE stands for *Legal Execution*, and is the sequence of steps such that the system continuously exists in well behaved states.

2.3 Evaluation Criteria

A common evaluation criteria in the field is to measure operation latency; the average time it takes for an operation to complete [18]. This includes both communication delay and local processing time. We use this as our primary metric for evaluation.

The operation latency is measured both in an isolated setting where no other clients are doing any requests and in a setting when we have different levels of base load on the servers. For comparison, we have two non self-stabilizing versions of CAS, as well as a MWMR implementation of ABD that uses a self stabilizing quorum system. The evaluation platform and the different test cases are described in Chapter 6.

3

Theoretical Background

In this chapter, we present the theoretical background on which our project is based. We go through core concepts and algorithms which has not been developed by us, yet plays an integral role in our project. We begin by explaining the concept of a quorum system and shared memory emulation, and proceed to detail the algorithms which are used in this project. The first algorithm is used as a basis for the communication channel, and is used for communication between nodes in our main algorithm. We then conclude by describing the two shared-memory emulation algorithms – ABD and CAS. ABD is used as a baseline comparison for performance, and CAS is of course at the core of the project.

3.1 Quorum Systems

Famously, highly distributed services stand before the issue of the CAP theorem – *consistency*, *availability* and *partition tolerance* are all important qualities, but achieving all three is not always possible. But as discussed in [19], it is not necessarily the case that only two out of the three characteristics can be reached. In fact, it is often the case that systems can deliver better than that, and using quorum based solutions is a wide-spread approach to do it.

A quorum is defined as the smallest subset of participants needed to make a decision. Exactly which subset that is, depends on the application. In a situation where the quorum size is equal to the number of participants in the quorum system, every participants needs to be consulted. Conversely, if the quorum size is one, only a single participant needs to be contacted (possibly because of one participant having an elevated position).

In systems with very large numbers of participants, it's exceedingly unlikely that everyone is available at a given time. If each server has an uptime of 99.99%, in a system with 10 000 servers that would mean only a 36% chance that all servers are available. And today it is common to have millions of servers in a cloud infrastructure [20, 21, 22], so it is clear that reasonable availability guarantees can not be given if all servers need to be contacted.

In applications which requires consistency in a partition free environment (e.g., reaching consensus), a quorum system can be designed to give such guarantees while providing superior availability compared to contacting all participants. As long as the definition of a quorum assures overlap between itself and any other quorum, consensus can be reached. The trivial solution to this is what is called a majority vote quorum. If responses were received from a majority of participants, it is impossible to construct another subset which comprises a majority of the participant without there being overlap. There are also other alternatives which can guarantee overlap, like for example a matrix based quorum system (where one row and one column in the matrix of servers are required to have a quorum). Formally, in a system of quorums $\mathcal{Q} = \{Q_1, Q_2, \dots\}$ it must hold that $\nexists Q_x, Q_y \in \mathcal{Q} : Q_x \cap Q_y = \emptyset$.

When designing a quorum system to provide great availability, it is generally desirable to define the quorum to be as small as possible. A small quorum naturally minimizes the number of servers which needs to be contacted, but in order to guarantee the desired functionality of an application it can usually not be arbitrarily small. In CAS for example, there are two essential attributes which both puts requirements on the quorum size: consistency and coding (see Section 3.5). It is not enough to simply require a majority (and thus ensuring overlap), but the overlap must be at least of size k. Otherwise, the original message can not be reconstructed from the coded elements. This lends another very interesting effect of the choice of coding parameter k: with less redundancy (and thus smaller code words) more responses are required. So while less redundant data being sent would mean strictly better performance on a traditional system, it can actually have negative effect in a quorum system since it would require more responses than otherwise.

Another advantage of quorum based solutions over contacting a predetermined set of participants, is that not only are fewer answers needed – it is in particular only the $|Q|$ fastest answers that are needed. In other words, we never have to wait for the slowest participants. This is especially useful in a heterogeneous setting such as for servers on the Internet, which may have widely varying load, different resources available or simply located at different places around the world. We

again want to point out that this benefit is forfeited if the coding is set to make the code words as small (i.e., non-redundant) as possible.

3.2 Shared Memory Emulation

The goal of emulating a shared memory is to hide the message passing from the clients, to instead provide the low-level operation primitives read and write. These operations are invoked from an external source, using a client as proxy. Operations on the memory should have the atomicity property and therefore appear as sequential.

3.2.1 Atomicity

There are two main criteria that need to be satisfied for the atomicity property. One is that any invocation of a read operation, after a write operation is completed, must return a value at least as recent as the value written by that write operation. The other is that a read operation that follows another read operation will return a value at least as recent as the value returned by the first read operation.

3.2.2 Shared Memory Model

Shared memory model is a cornerstone of distributed computing. The setup is that nodes communicate with each other by doing read and write operations to a physically shared memory. The memory is often either single-writer and multiple-reader (SWMR) or multiple-writer and multiple-reader (MWMR).

In shared memory, it is an unreliable and asynchronous setting where nodes can fail-stop is assumed. Due to spatial locality in shared memory, it is easier to get an overall perspective on the system, compared to object sharing in message passing systems.

3.2.3 Object Sharing in Message Passing Systems

Shared memory makes it fast and easy to share data objects between processes. But doing so on an asynchronous message passing platform is much harder, since there might be link failures and unbounded communication delays. Yet, it is vital that systems can share data reliably with each other on the Internet.

Objects can be shared in a message passing system by emulating a shared memory. An object is accessed using a series of operations at a client node in order to satisfy the atomicity property. But links can fail and therefore can information be temporarily unavailable due to partitioning.

3.3 Communication Channel

A self-stabilizing communication channel is needed for Algorithm 3 in [6]. A channel is constructed using the self-stabilizing version of the token passing algorithm described in [17, Figure 4.1]. The extension needed to make that algorithm self-stabilizing is simply to increment the sender's counter modulo $cap + 1$, where *cap* is the upper bound on the number of messages that can be in transit (the *channel capacity*). It is assumed that the number of latent messages in the channel is less than 2^{32}, which we deem to be sufficiently large. The pseudocode for the self-stabilizing communication channel can be found in Algorithm 1 and Algorithm 2.

3.3.1 Token Passing Algorithm

In order to make Dolev's self-stabilizing communication channel realizable, without a mechanism for removing old messages, a slight modification is necessary. At line 11 in [17, Figure 4.1], there is an else-statement: `else send(counter)`. That line can cause practical implementation issues. In the (presumably relatively rare) event of the receiving end being slower than usual, the timeout might trigger and cause one or many retransmissions of the counter. There would thus be at least two tokens in circulation, assuming the original message was slow but never dropped. If the sender *also* resends upon any message arrival (which the aforementioned line 11 would bring about), the channel would never have cause to get the extra tokens out of the system. Our modification does not change the theory, since the

timeout *will* eventually trigger a retransmission if the if-statement is never entered.

There are other self-stabilizing communication channels that could be used instead, like [23] or [24]. The advantage of implementing a channel based on a token passing algorithm is that it uses a stop-and-wait approach, while [23] and [24] relies on a repeating retransmission of messages until it receives an acknowledgement. That way, an application layer mechanism for congestion control does not have to be implemented since it is not needed in a stop-and-wait communication.

3.3.2 Sender Algorithm

A sender has access to three local variables: *message*, *counter* and *cap*. A message that is about to be sent is placed in *message*. The current value of the token is held by *counter* and *cap* is the upper bound on the number of messages that can exist simultaneously in the channel.

The sender protocol has two types of events (see Algorithm 1). The first event (line 1) is a timeout on the communication between processor i and j. This functionality is necessary in order to retransmit, in case a message or the token is lost. The second event (line 3) is triggered whenever a message arrives at processor i. When such a message arrives the message counter field is examined to see if it is a fresh message with an up-to-date token. If so, the local counter is updated and a token arrival event is triggered. This event could be either an acknowledgement from a gossip message or a response to a PingPong request. The data sent is a concatenation of the token and the message and once it is invoked, the sender stops holding the token.

Algorithm 1: Algorithm for the sender's protocol in the communication channel.

Variables: $message, counter, cap$

1 **upon** *timeout on message from* p_j *to* p_i **do**
2 send$(j, counter||message)$
3 **upon** $(msgCounter, message)$ *from* p_j *to* p_i **do**
4 **if** $msgCounter \geq counter$ **then**
5 $counter \longleftarrow (msgCounter + 1)\ \%\ cap$
6 **raise** *pingpong* or *gossip* event according to channel type
7 send$(j, counter||message)$

3.3.3 Receiver Algorithm

A receiver has access to the local variables *counter* and *response*. The latest non-duplicated token value is stored in *counter* and the response (whether it is an acknowledgment or not) in *response*.

The receiver algorithm has only one event and it is triggered on every message arrival (see Algorithm 2). The received token is examined and compared to the local counter value. If they are different, then the receiver has the token and can trigger a second event accordingly. The counter together with a response is then sent back to the sender.

Algorithm 2: Algorithm for the receiver's protocol in the communication channel.

Variables: $counter, response$

8 **upon** $msgCounter, message$ *from* p_j *to* p_i **do**
9 **if** $msgCounter \neq counter$ **then**
10 $counter \longleftarrow msgCounter$
11 **raise** *pingpong* event or $response \leftarrow ACK$ according to channel type
12 send($counter||response$)

3.4 ABD

One of the first and most significant contributions to shared memory emulation algorithms is the work by Attiya, Bar-Noy and Dolev [3], which is most known as the *ABD* algorithm. The ABD algorithm has been iterated upon by Lynch and Shvartsman [4], making it MWMR. A similar MWMR implementation using general quorums is the Simple algorithm proposed by Nicolaou and Georgiou [10]. Georgiou has created a summary of Simple, with pseudo code, in [25].

ABD is a SWMR algorithm, and employs full data replication with a majority vote quorum system to achieve data consistency. One of the key elements introduced in [3] was the use of so-called tags, for versioning of the data objects. In the SWMR case the tag can simply be a counter, incremented by the writer each time a new object is written. But a tag might also include the *unique identifier* (UID) of the writer client, which is needed in the MWMR versions of the algorithm. Each tag

is associated with some data value, `<timestamp, value>`, and is (together with majority vote quorums) the key to consistency in ABD.

ABD uses full replication, which results in the total amount of data stored being quite large. Denoting the number of servers N and the size of one data object d, the system will need to store a total of $N \times d$. This means that a single write from a single client could occupy quite considerable resources, both on the network links and storage wise.

3.4.1 ABD Client Algorithm

On the client side, ABD has two distinct parts: a read protocol and a write protocol. As per the MWMR version presented in [25], both protocols consists of two phases. For a writer client, we call the first phase the *writer query* and the second phase the *write phase*. During the writer query, the client sends a *query message* to all servers, requesting their respective latest tag. The client then waits until it has received responses from a quorum of servers, and stores the greatest tag value as $maxTag$. The writer query phase now ends, and the write phase can commence. The client increments the tag counter to be $maxTag + 1$, adds its own identifier to the tag, and then proceeds by sending a message (containing the new tag) to all servers. After receiving acknowledgements from a quorum of servers, the client is assured to have successfully written a new value to the quorum system.

A reader client has two phases, and much like a writer client it starts with a *query phase*. The reader query requests the latest timestamp from every server, and waits until is has received a quorum of responses containing tuples of tag and value. It then finds the greatest tag amongst the received responses, and stores it as $maxTag$. After the query phase the client enters the *propagation phase*, in which it disseminates its knowledge of the globally most recent tag value found. The client sends out $maxTag$ to all servers, and subsequently waits for acknowledgements from a quorum of servers. At this point, the read operation is considered successful (returning the value corresponding to $maxTag$).

3.5 Coded Atomic Storage

Coded Atomic Storage (CAS) was presented by Cadambe et al. in [5]. It builds on techniques which were introduced already in 2003 by Fan and Lynch [11] and Lynch and Shvartsman [26], and provides an efficient MWMR algorithm for shared memory emulation. It uses erasure coding to reduce the communication cost, instead of using full replication like ABD. CAS is a quorum based algorithm, where a quorum is any subset Q of all servers such that $|Q| = \lceil \frac{N+k}{2} \rceil$. N is the number of servers and k is the coding parameter deciding how many elements are needed to reassemble the message. The CAS algorithm allows for up to f server failures.

3.5.1 Erasure Codes

Erasure coding is a technique whereby a relatively small amount of redundant information is added to a piece of data, in order to make it robust to bit erasures. An (N, k) erasure code splits the data into N coded elements which has coding applied to them such that only a subset containing k elements is needed to reassemble the original data. An erasure code is said to be a *maximum distance separable* (MDS) code if it has the property that the original data can be reconstructed from any k of the N coded elements (as opposed to requiring one or more of the k elements to be of a particular kind). The particular kind of MDS erasure coding we use is (N, k)-Reed-Solomon codes, which is a group of MDS erasure codes.

CAS builds on having (N, k) coding applied to the data, and distributing the N coded elements to the servers in the quorum system. Since k elements are required in order to reassemble the data, the coding parameters have a direct effect on the quorum size. This accommodates for a flexibility in choosing between having smaller sized coded elements and better data redundancy – CAS can be tweaked according to the system needs. The fraction $r = k/N$ is called the code rate, and is a measure of how big part of the coded elements is non-redundant data. A high code rate means less redundancy but smaller elements, while the inverse is true for low code rates. The special case of $k = 1$ is effectively equivalent to full replication.

3.5.2 CAS Client Algorithm

All algorithms for shared memory emulation provides read and write operations for the clients. The writer's protocol in CAS has three phases: a *query*, a *pre-write* and a *finalize* phase. The reader's protocol in CAS has only two phases, a *query* and a *finalize* phase. The phases in the respective protocols are described below.

CAS Writer's Protocol

Query sends a message to all servers to request the highest tag that has the label finalized. The client then waits until it has received responses from at least a quorum of servers. It then takes the maximum of all received tags, which is the global maximum tag.

Pre-write sends a message on the form (t, m, 'pre') to all servers and waits for an acknowledgement from a quorum of servers. The variable t is the received tag-number from the query phase, increased by one. The variable m holds a coded element (see Section 3.5.1).

Finalize sends a message (t, $\perp$, 'wfin') to all servers. After receiving a quorum of acknowledgements, the write operation is finished. The finalize phase hides the write operations that have not been seen by a quorum, since the query phase only looks at records with phase 'fin'. Once the client has passed the *pre-write* phase, it knows that at least a whole quorum has enough elements to reconstruct the data and therefore it can be made visible in other operations.

CAS Reader's Protocol

Query: this phase is identical to the query phase in the writer's protocol.

Finalize sends out a message (t, $\perp$, 'rfin') to all servers, where t is the maximum tag-number calculated during the query phase. The label 'rfin' is used for the server do differentiate between a reader finalize and a writer finalize. The client waits until every server in a quorum has responded with the coded element corresponding to t (if they have it) or an empty object. If at least k of the servers that responded have included a coded element, the reader will reassemble the message and return it to the application. Otherwise, it just returns as an unsuccessful read.

3.5.3 CAS Server

A server has a local state that is its storage of records. A record has the form $(t, w, label)$ where t is a tag, w is a coded element and $label$ is either 'pre' or 'fin'. A tag is a tuple of sequence number and an identifier for the client that originally created the tag. The server protocol also has event handlers corresponding to the client requests: query, pre-write, read-finalize and write-finalize.

Clients can fail-stop during a write operation, and there might exist unfinished records because of non-finalized records. If a client crashes during the pre-write phase, it is not a problem for other clients since this record is not yet visible to them. However, if a client crashes during the finalize phase, then that tag is visible during another client's query phase, even if it was not finalized on an entire quorum. This potential problem is solved by having every server periodically gossip their finalized records to the other servers. This mechanism is used to implicitly finalize records which has already been finalized by other servers.

4

Algorithms

In this chapter, we introduce the algorithms which has been developed during this project. The first section describes the reincarnation service, which provides a way for clients to resume with a unique identifier after a crash. In the section following that, we describe the mechanism for global reset. The algorithm used for global reset builds on principles from [1], but was adapted to fit our purposes.

4.1 Reincarnation Service

Reincarnation of clients is an extension mentioned in Dolev et al. [6]. Without this extension, clients must never resume after a crash, except as part of the global reset procedure. If a client sends a request, crashes and then restarts again and performs a new request, the response to the first request might be received and mistaken as the response to the new request. That would violate correctness, and is dealt with by the reincarnation service.

4.1.1 Client Identifier

A client identifier consists of an incarnation number and a unique hardware address. The incarnation number is requested (and updated, if needed) at boot as well as periodically, in order to ensure that it is updated after a crash. This way, clients do not have to delay joining until a global reset is triggered by a transient fault, but can resume immediately. However, this will put a new bound on the number of clients allowed in the system. It is not only the amount of client nodes, but the number of client nodes and their incarnations.

It is important to note that a new type of sequence number (the *incarnation number*) has been introduced and it too is prone to transient faults. Such a situation is handled the same way as with a maximum tag number. If a maximum incarnation number is noticed, then new queries are blocked and a global reset is invoked.

4.1.2 Reincarnation Service Algorithm

The pseudocode for the reincarnation service algorithm can be found in Algorithm 3. The algorithm includes both server and client functionality as well as their local variables.

A client has access to the *uid* variable, which is a tuple of incarnation number and the client's globally unique hardware address. The client also has access to an interface called $qrmAccess()$, which gives access to a majority quorum of servers. A server has a first-in-first-out queue where it stores tuples containing the hardware address and highest corresponding incarnation number for each client. In order to bound the storage space it is assumed that there exists an upper bound on the space of relevant hardware addresses. However, since it is a queue, the set of relevant addresses can vary over time.

The client algorithm performs a periodic task which starts with a query phase to check if its current incarnation number is up to date. It queries all servers, and awaits responses form a quorum of servers. The maximum value of all received incarnation numbers is calculated, and if that number differs from the current client incarnation number, a second phase is triggered. In the second phase, the incarnation number is updated both at the client side and in the quorum system. The client takes the maximum of the current incarnation number and all received incarnation numbers, increments that by one and sends it out to all servers. After receiving a quorum of acknowledgements, the client knows that it has been assigned a new valid incarnation number and can thus proceed with operation as usual by updating its *uid* accordingly.

The server algorithm has two event types that can be triggered: a request for a incarnation number and an update of an old value. The query procedure first checks that the maximum allowed incarnation number does not appear in the server's incarnation number queue. If there exists such a value, new incarnation number requests will be blocked in the query phase until a global reset has completed. Otherwise, if no previous number associated with the requested hardware address

exists, the default value 0 is returned. If the client's previous number is present, then that tuple is placed at the tail of the queue and is sent as a response to the request. The update procedure is simpler and just adds the new value to the queue. If a previous value was recorded, then the update procedure removes the old value from the queue.

Algorithm 3: Algorithm for reincarnation service.

Variables:
uid: is a tuple of hardware address and incarnation number
cntrs: is a FIFO queue of all incarnation numbers associated with a corresponding hardware address. The size of the queue is the upper bound on the number of relevant hardware addresses allowed. New entries are included in the queue after one complete cycle.

```
1  The client:
2  upon periodic task do
3      let incNbr ⟵ max{qrmAccess('cntrQry')}
4      if incNbr ≠ uid.incNbr then
5          newIncNbr ← max{incNbr, uid.incNbr} + 1
6          qrmAccess((newIncNbr,'incCntr'))
7          uid ⟵ ⟨hwAddr, newIncNbr⟩
8  The server:
9  upon cntrQry arrival from p_j's client to p_i's server do
10     if maxIncNbr ∈ cntrs then return
11     if ⟨j, •⟩ ∈ cntrs then
12         cntrs.add(cntrs.remove(j))
13         reply(j, (cntrs.get(j).incNbr,'cntrQry'))
14     else
15         reply(j, (0,'cntrQry'))
16 upon (newIncNbr,'incCntr') arrival from p_j's client to p_i's server do
17     if ⟨j, •⟩ ∈ cntrs then cntrs.remove(j)
18     cntrs.add(⟨hwAddr_j, newIncNbr⟩)
```

4.2 Global Reset

A global reset mechanism is needed to reset sequence numbers and wrap around to a default value. That a sequence number reaches its maximum value is in

practice only possible due to a transient fault. In order to wrap around, all servers have to be alive for agreement to happen. Guaranteeing that every server in a configuration is alive requires access to reliable failure detectors. As this is not the main focus of this project, we assume that an external mechanism is implemented to remove failed servers from the configuration [1].

4.2.1 Global Reset Algorithm

The ideas behind the Global Reset Algorithm is similar to [1, Algorithm 3.1] by Dolev et al. Their algorithm is a self-stabilizing membership reconfiguration but has been adapted to act as a reset mechanism in our context.

The key idea is that a server proposes a tag and then a coordinated phase transition used in order for all servers to agree on that specific tag number. When they have reached an agreement, all other tag numbers should be removed from their storage (in the $localReset()$ procedure). This is referred to as the replacement phase. Since this is a self-stabilizing algorithm, it constantly checks for transient faults. If a transient fault is noticed, the algorithm cancels the replacement phase and enters a reset phase. This reset phase is used to restart the replacement process. The pseudocode for the adapted algorithm can be found in Algorithm 4. This requires an updated version of the gossip protocol in Dolev et al. [6] which can be found in Algorithm 5.

Algorithm 4 requires the local variables: prp, all, $echoAnswers$, $allSeenProcessors$, $config$ and $dfltPrp$. The lists prp and all store received proposals and whether or not all have seen their proposals respectively. The list $echoAnswers$ holds the latest value that a processor has sent, which has also been acknowledged by the servers. The set $allSeenProcessors$ are used to gather all servers that have reported that everyone has seen their proposal. Addresses to all participants are in $config$ and $dfltNtf$ is a default proposal used when there is no wrap around currently in progress. The proof, and how the mechanism works, can be found in the original paper [1].

4.2.2 Reset Indication

A reset of the proposals is done when a transient fault is detected. This process is triggered by line 22 and results in a $\perp$ in every $prp_i[k] : p_k \in config$. The goal of

this is to stop any ongoing global reset procedure and start over from a state where every processor p_i has $prp_i[i] = dfltPrp$. During the reset phase no processor can propose a new record with a call to $propose(tag)$ due to being blocked by the macro $enableReset()$ until the reset phase has finished.

Line 21 should catch every possible transient fault. The arguments are similar to the paper by Dolev et al. [1], but are adopted for our purpose. One of the main differences is that we do not make use of failure detectors but rather assume that there exists another mechanism to ensure that every processor will be alive both during the reset phase as well as the replacement phase. Another difference is that both the replacement mechanism and the reset mechanism from [1] works with proposals for a record with the current highest tag number and not a new configuration. Therefore the trigger is when there exists any processor or message with its proposal field set to $\perp$. In the original paper, Lemma 3.2 is triggered by conflicting configurations and this can be due to transient faults or servers restarting with a different local state than other servers.

Algorithm 4: Algorithm to perform a global reset, using coordinated phase transitions.

Variables:
$prp[]$, $all[]$, $echoAnswers[]$, $allSeenProcessors$, $dfltPrp = \langle 0, \bot \rangle$
function: $propose(tag) =$ {**if** $enableReset()$ **then**
$(prp[i], all[i]) \leftarrow (\langle 1, tag \rangle, false)$}
macro: $enableReset() =$ **return**$(\nexists p_k \in config : (prp[k] = \bot)$
$\vee ((prp[k], all[k]) \neq (dfltPrp, true)))$
macro: $prpSet(val) =$ **foreach** $p_k \in config$ **do** $(prp[k], all[k]) \leftarrow (val, false)$
macro: $modMax() =$ **if** $Phs = \{0, 1\}$ **then return** $\max Phs$ **else return**
$prp[i].phase$, **where** $Phs = \{prp[k].phase\}_{p_k \in config}$
macro: $degree(k) =$ **return** $(2 \cdot prp[k].phase + |\{1 : myAll(k)\}|)$
macro: $corrDeg(k, k') =$ **return** $(\{\{degree(k), degree(k')\} \in$
$\{\{x, x\}, \{x, x + 1 \bmod 6\}, \{x, x + 2 \bmod 6\} : x \in \{0, \ldots, 5\}\})$
macro: $maxPrp() =$ **if** $\{(degree(k) - degree(i)) \bmod 6\}_{p_k \in config} \nsubseteq \{0, 1\}$
then return $prp[i]$ **else return**
$\langle modMax(), max_{lex}\{prp[k].tag\}_{p_k \in config} \rangle$
macro: $myAll(k) =$ **return** $(all[k] \vee (\exists p_l \in allSeenProcessors :$
$prp[i].phase + 1 \bmod 3 = prp[l].phase))$
macro: $greaterOrEqual(k) =$ **return**
$(prp[i].phase + 1) \bmod 3 = prp[k].phase \vee prp[i] = prp[k]$
macro: $echoNoAll(k) =$ **return**
$(prp[i] = echoAnswers[k].prp) \wedge greaterOrEqual(k)$
macro: $echo(k) =$ **return**
$(\{(prp[i], all[i])\} = \{echoAnswers[k]\}) \wedge greaterOrEqual(k)$
macro: $increment(prp) =$ **case** $(prp.phase)$ **of** 1**: return**
$(\langle 2, prp.tag \rangle, false)$**; 2: return** $(dfltPrp, false)$**; else return**
$(prp[i], all[i])$**;**
macro: $allSeen() = (all[i] \wedge config \subseteq (allSeenProcessors \cup \{p_i\}))$
macro: $proposalSet = \{prp[k].tag : \exists p_{k'} \in config : prp[k'] = \langle 2, \bullet \rangle\}_{p_k \in config}$
Do forever begin
  foreach $p_k \in config : all[k]$ **do**
  $allSeenProcessors \leftarrow allSeenProcessors \cup \{p_k\}$
  if $(\exists p_k : ((prp[k] = \langle 0, s \rangle) \wedge (s \neq \bot)) \vee (\exists p_k, p_{k'} \in config :$
  $\neg corrDeg(k, k')) \vee (\{p_k \in config : prp[i].phase + 1 \bmod 3 =$
  $prp[k].phase\} \nsubseteq allSeenProcessors) \vee (|proposalSet| > 1) \vee ((\exists p_k \in$
  $config : prp[k] = \bot) \wedge (prp[i] \neq \{dfltPrp\})))$ **then**
    $prpSet(\bot)$
  if $(prp[i] = \bot \wedge all[i])$ **then**
    $prp[i] \leftarrow dfltPrp$
  $(prp[i], all[i]) \leftarrow (maxPrp(), \bigwedge_{p_k \in config}(echoNoAll(k)))$
  if $(Prps \neq \{dfltPrp\} \wedge \nexists x \in Prps : x = \bot)$, *where*
  $Prps = \{prp[k]\}_{p_k \in config}$ **then**
    if $allSeen() \wedge \bigwedge_{p_k \in config}(echo(k))$ **then**
    $((prp[i], all[i]), allSeenProcessors) \leftarrow (increment(prp[i]), \emptyset)$
    if $prp[i].phase = 2$ **then** $localReset(prp[i].tag)$

5

Implementation

In this chapter, we describe the system implementation. We begin with a short mention of our development approach, where we discuss the programming language, paradigms, external libraries as well as testing approach. We then proceed to a more concrete discussion regarding the implementation choices and outcome. The first concrete part is the communication channel, which is discussed in Section 5.2. In Section 5.3, the Gossip and PingPong protocols are described. Finally, the actual implementation outcome of ABD and CAS is described.

5.1 Approach

This section first describes the choice of development method, and then how continuous testing was conducted, as well as the test bed. The code developed during this project is licensed under the MIT license, and can be found in our code repository[1].

5.1.1 Development

The library is developed using the Python programming language. It is a good general purpose programming language that allows fast prototyping and, since it is an interpreted language, makes the library portable.

We chose to work in the object oriented programming paradigm, using an asyn-

[1] `https://bitbucket.org/selfStabilizingAtomicStorage/datx05-code`

chronous, event-driven approach to concurrency using the Asyncio library. Asyncio can be used to write single-threaded concurrent code. When programming with Asyncio, one does not run code explicitly. Instead, functions (or coroutines) are scheduled in an event loop. The two main reasons for using asynchronous programming is less overhead compared to spawning new threads and the code structure makes it easier to argue about its correctness. We want the software to be event driven, since that ties well into both the asynchronous nature of the system and how the algorithm in [6] is described.

Some of the necessary functionality is already implemented and therefore it is used as dependency instead of reimplementing it. One of the libraries is a cross-platform library called *liberasurecode* that provides implementations of Reed-Solomon encoding and decoding. This library can be accessed from a Python application using the PyEClib[2] library, which serves as an interface to liberasurecode.

Python's own socket class, which is an abstraction of a low-level networking interface is used to create network endpoints. Asyncio, as part of the standard library, provide non-blocking socket I/O operations. Another blocking operation is file I/O. Reading and writing of large files to disk takes in relative terms substantial time so asynchronous file operations are handled using Aiofiles[3].

5.1.2 Testing

To test the implementation during development, the discrete event simulator NS-3 is used to simulate a real LAN environment. This makes the testing more realistic because of the possibility of changing network configurations.

The *NS* family of simulators is widely used in research and the latest version NS-3 has been shown to have good performance compared to other network simulators [27]. The controlled test environment consists of a simulated network where we can control the communication latency and the data rate, similar to the approach taken in a previous master thesis project at Chalmers [16] or as described in the official NS Wiki [28].

In order to standardize the test environment in which the implementation is run, we are using containerisation software to isolate each instance of the application.

[2] `https://github.com/openstack/pyeclib`
[3] `https://pypi.python.org/pypi/aiofiles`

Containers can be very light-weight compared to virtual machines, while still providing isolation from the host system. This way, together with the simulated network, we get a full-stack environment that acts as if there were multiple different machines running, connected via a network. The Docker ecosystem makes it easy to setup automated test environments, which in turn can be deployed on any machine with Docker installed.

5.2 Communication Channel

The communication channel consist of two parts, a sender and a receiver. The sender is responsible for reliable transfer of a message to the receiver. The channel is constantly circulating a token between the sender and the receiver. Upon a token arrival at the sender, a new message is put in the send buffer. Upon token arrival at the server, a response is calculated and brought back to the sender along with the token. A token is an integer and is prepended to the payload.

5.2.1 Dual Transport Protocols

TCP is not a self-stabilizing protocol, but it has many advantages over UDP. TCP has an assortment of nifty features built in to maximize throughput while treating other connections fairly and also not overloading the receiving party. UDP does not have that, and implementing it in a higher level layer is not trivial.

One can avoid the problem altogether by never sending more often than one packet every round trip time (RTT). Then the only time when multiple transmissions without intermittent ACK could occur is during retransmission of datagrams. Some algorithms use a constant retransmission strategy (*"send infinitely often"*), while those which use a stop-and-wait strategy must handle omission faults by retransmitting after a timeout when no ACK was received. According to RFC 8085 [29], an application using UDP which does not receive return traffic should send no more than one UDP datagram every 3 seconds (in order to avoid congestion). Since the stop-and-wait strategy necessarily waits for at least 1 RTT before retransmitting (assuming a reasonably long timeout) it needs no further congestion control. The only control mechanism needed for a stop-and-wait based channel is to have a fair retransmission strategy for when no ACK is received.

One issue with sending only once every RTT is that it can be prohibitively slow for large payloads. Since one of the main advantages of CAS is the erasure coding, it should be beneficial to write large data objects. When sending large data objects, the 2^{16} bytes of data that fits in a UDP datagram might not be enough. To transfer objects larger than that, a TCP connection is established where both the token and data object is transferred. After the transfer is finished the connection is torn down. Due to only a single token is sent, the behaviour of sending a single message (even though it is actually an entire TCP session) is emulated. An illustration of how that switch between UDP and TCP would look like (from the perspective of the communication channel) is found in Figure 5.1.

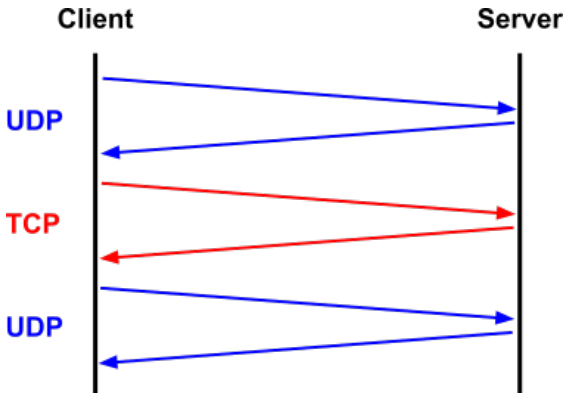


Figure 5.1: High-level view of how the switch between UDP and TCP should be perceived by the channel, when transferring a large data object. The red arrows marking the TCP part represents an entire TCP session.

5.2.2 Sender Channel

A sender channel is used to communicate with only one peer and is therefore associated with a specific peer address and a channel type. The peer address is IP and port number of a specific server end point. The channel type specifies what type of message is exchanged in the channel and is needed in order to determine the protocol for the communication. A message channel indicator is put before the actual protocol data, together with the token.

With the sender channel it is possible to switch between TCP and UDP between communication rounds. Which one to use is determined by setting a specific flag in the channel. The starting state of a channel is only passing a token back and forth and is therefore initialized to use UDP. When to change to TCP and back are determined by the quorum and gossip service, which is explained in Section 5.2.1.

5.2.3 Server Receiver Channel

A receiver channel is used by a server in order for sender channels to connect. A server handles many connected sender channels at the same time and are not aware beforehand how many will exist. The first time a sender channel connects and sends a message, the server will do a lookup in a table to see if there is a previous token associated with the sender. If that is the case then the new token will be compared to the previous received one, otherwise it will be added to the table. If the token indicates that this is a fresh message, then it will be further processed by looking at the message type and parse it with the correct protocol.

The server is serving both TCP and UDP requests and the response is sent using the same transport protocol as the request. A UDP request only consists of a single message and is therefore straight forward to receive. Since a TCP request is a stream of bytes it is more difficult to determine when and when not to expect more data. This is solved by first sending four bytes that contain the total size of the complete message. The receiver end can use this information to know how much more data to expect and when to finish the receiving part and start process the data.

5.2.4 Multiplexing

As described in Section 5.1.1, we make use of the asyncio library for multiplexing instead of the more traditional approach of using threads. When using threads, the context switching is already dealt with at operating system level, but that does not happen when doing concurrency at the code level. The downside of spawning new threads for each connections is that it will exhaust system resources quickly. Another approach is to use a thread pool, but that risks ending up in a deadlock situation unless much attention is devoted to managing mutual exclusion properly.

Using asynchronous programming, the program itself is responsible for yielding processor time. So when sending or receiving fairly large data objects over TCP, this might block other concurrent events. To overcome this and reduce the overall response time, we define a parameter which specifies a threshold on a TCP transfer. If this threshold is reached, the sending or receiving is rescheduled in order to let other events run. This way, each client will get a piece of the information at a time, in a round robin fashion.

5.3 Communication Protocols

In order to interpret the byte stream that is passed from one network socket to another, it needs to follow a certain structure. If all the participants in the communication knows the underlying structure, they can easily parse the stream and read the data. The received data is parsed according to a protocol that is associated with the channel. The two different protocols are the PingPong protocol and the Gossip protocol.

5.3.1 Gossip Protocol

The Gossip protocol is used in the communication between all servers. This is to exchange information to eventually update stale information at servers. A Gossip message contain the three highest tag numbers with the labels *pre, fin, FIN* (see *tagTuple* in Chapter 3), that is used for the correctness of the self-stabilizing CAS implementation. Along with the tag numbers are information about the global reset mechanism and the set of incarnation numbers exchanged.

How often gossip messages are sent will have an impact on the performance of client operations. How much it will affect depends most on the overall load on the servers. Things that should be considered is how often should servers send their gossip messages and whether or not all information have to be included in every message. Such configuration parameters should not be "hard coded" since what setup to prefer might change depending on where it is used. In order to make these parameters easy to adjust, they are part of a start-up configuration file that the server is initialized with during boot up procedure.

5.3.2 PingPong Protocol

The PingPong protocol is used between a client sender and a server receiver. This protocol is used in Dolev et al. [6] and was originally described as part of the *communicate* procedure in Attiya et al. [3]. The implementation of this protocol contain the fields: label, tag, mode, payload and a request tag. Label is used to determine the operation. A tag is used as reference to a specific data object. Mode is used to determine if it is part of a read or write operation. Payload is the byte element that is a part of a data object. In order to match the response with the

right request, the request is always passed back with the server response.

A PingPong message is an object created at a client node as part of a quorum access procedure and is later converted to a byte stream before delivered to the transport protocol. A server receives the byte object and if it is identified as a PingPong protocol is is parsed accordingly. The server looks at the message, triggers an event as part of the response and then passes it back to the client piggy-backed with the token.

5.4 Implemented algorithms

In addition the main algorithm, which is the self-stabilising CAS, we have also implemented two other shared memory algorithms. The other algorithms are a MWMR version of ABD and a CAS version without self-stabilising mechanisms.

5.4.1 MWMR ABD

The MWMR ABD implementation is based on the Simple algorithm proposed by Nicolaou and Georgiou [10], which is described in Section 3.4. This algorithm is implemented on top of the robust communication channel in Section 5.2. The algorithm itself does not require a self-stabilizing communication channel, because the algorithm is not self-stabilising, but it is good for a fair comparison during the performance evaluation. By using the communication channel we can look at differences in the algorithms rather than comparing implementation specific details.

5.4.2 CAS Algorithm

To measure the overhead caused by the self-stabilising mechanisms in CASSS, we have a previously implemented version of the original CAS algorithm. This implementation is quite different from the self-stabilising version. The main difference is that it does not use the communication channel described in Section 5.2, but instead uses the ZeroMQ[4] library to send and receive atomic messages.

[4] `https://zeromq.org`

ZeroMQ is a messaging library that abstracts traditional sockets and solves many of the networking issues that need to be handled otherwise. This means that a TCP session can be started during the initialization of the program and any disconnection or lost messages is dealt with automatically in the background. This gives a performance advantage over the implementation of the self-stabilizing CAS.

5.4.3 Self-Stabilizing CAS

The only difference between the client side protocol of the self-stabilizing version of CAS and the traditional CAS algorithm is that the write operation has one more round. On the server side, the main differences are the updated gossip protocol (see Algorithm 5), and of course updated event handlers for the clients' requests. Common to both the client side and server side is that all the building blocks needs to be self-stabilizing. More details, along with pseudo code, can be found in [6].

The algorithm uses the communication channel described in Section 5.2 and the communication protocol described in Section 5.3. The integration of functionalities is done according to Algorithm 5. Algorithm 5 is an updated version of the gossip protocol presented by [6]. In addition to the original version, this includes both testing for transient faults and keeping the table of incarnation numbers updated.

Algorithm 5 has several variables, macros and functions that are defined in other algorithms (Algorithm 2 in [6], Algorithm 3, Algorithm 4) but are present here for integration purposes. A macro called *stabilize* is defined here, which tests whether or not all servers report the same set of tags and the maximal tags have propagated.

The first if-statement is for establishing a new proposal if a maximum integer has been seen in the register. Otherwise, the records are updated according to the gossip messages and irrelevant records are removed. Line 35–37 updates incarnation numbers and in line 38 is a new gossip sent.

Algorithm 5: Algorithm for the gossip protocol.

macro: $stabilized() = (\forall p_k \in config[i] : gossip[k] = tagTuple()) \wedge (tagTuple() = (t, t', t') \wedge t \geq t')$

29 **upon** *gossip* $(\{(pre[k], fin[k], FIN[k]) = gossip[k]\}_{p_k \in P}, prp[j], all[j], echo[j], cntrs_j)$ **from** p_j **do**

30 **if** $(maxPhase(D) \geq t_{top} \vee maxIncNbr \in cntrs) \wedge stabilized()$ **then**

31 $propose(maxPhase(D \setminus \{\text{`}pre\text{'}\}))$

32 **else**

33 Line 59–64 from Dolev et al. [6]

34 $S \leftarrow relevant(S)$

35 **for** $\langle hwAddr, incNbr \rangle \in cntrs_j$ **do**

36 **if** $hwAddr \in cntrs_i$ **then** $currentIncNbr = cntrs_i.get(hwAddr)$ **else** $currentIncNbr = 0$

37 $cntrs_i.update(hwAddr, \max(incNbr, currentIncNbr))$

38 $gossip(tagTuple(), prp[i], all[i], (prp[j], echo[j]), cntrs_i)$

6

Evaluation Environment

In this chapter, we describe the evaluation environment. First, we describe the evaluation platform and its characteristics. After that, we describe the experiments used to evaluate the performance, and the rationale behind each of them.

6.1 Evaluation Platform

This section begins with a description of the PlanetLab platform, and the specific setup which was used as evaluation platform for this project. We then proceed by describing the particular experiments used for the evaluation.

6.1.1 PlanetLab

Experiments are conducted in a real-world scenario to evaluate the system performance. For this purpose we have access to the PlanetLab EU[1] platform, which provides us access to a set of virtual machines running Fedora 25. A PlanetLab user gets access to a containerised instance via Linux containers (LXC). The PlanetLab EU servers are distributed all over Europe, and since they are connected over the Internet, they do indeed provide a suitable environment to evaluate a real-world distributed application. Because of this, an application on PlanetLab has to deal with all the real-world issues one usually runs into, such as congestion, link failures and node failures. This makes it a good platform to evaluate the robustness of a distributed system.

[1] `https://www.planet-lab.eu/`

6.1.2 PlanetLab Setup

Table 6.1 lists the PlanetLab nodes which were used as servers, and Table 6.2 lists the PlanetLab nodes which were used as clients. Even though there are hundreds of machines available on the Planet Lab platform and they run the same operating system, they do differ in compatibility. We had to carefully pick nodes so that they had a global static IP, that applications were able to bind to ports and, for the case of client nodes, had the hardware support needed for the erasure coding library.

Hostname	**TLD**	**IP Address**
cse-yellow.cse.chalmers.se	se	129.16.20.70
planetlab-1.ing.unimo.it	it	155.185.54.249
planetlab-2.cs.ucy.ac.cy	cy	194.42.17.164
planetlab-2.ing.unimo.it	it	155.185.54.250
planetlab2.upm.ro	ro	193.226.19.31
ple1.cesnet.cz	cz	195.113.161.13
ple1.planet-lab.eu	eu	132.227.123.11
ple2.planet-lab.eu	eu	132.227.123.12
ple4.planet-lab.eu	eu	132.227.123.14
ple44.planet-lab.eu	eu	132.227.123.44

Table 6.1: The ten PlanetLab nodes which were used for servers in the experiments.

Hostname	**TLD**	**IP Address**
pl1.uni-rostock.de	de	139.30.241.191
pl2.uni-rostock.de	de	139.30.241.192
planet4.cs.huji.ac.il	il	132.65.240.103
planetlab11.net.in.tum.de	de	138.246.253.11
planetlab13.net.in.tum.de	de	138.246.253.13

Table 6.2: The five PlanetLab nodes which were used for clients in the experiments.

6.2 Experiment Scenarios

In this section, we describe the scenarios we constructed, and the underlying rationale for why we run the experiments we do. We begin by describing the generic

setting used for the experiments, and how the performance is measured. We then dive into the details of each scenario, and how they differ from the generic setting.

6.2.1 Baseline Settings

In order to standardise the evaluation setting, a baseline as presented below is used for each of the experiments unless otherwise noted. The configuration used for each specific experiment can be found in Appendix A.

The setting which all experiments proceed from is to have 15 machines in total, ten of which run one server process each and five of which run one client process each. When increasing the number of clients or servers beyond the amount of physical machines, multiple instances are put on the same physical machine. In order to guarantee a fair latency between a client and a server instance, clients processes are never placed on the same physical machine as server processes. More clients or servers than available nodes are distributed in a round-robin fashion. Operations of a client are invoked sequentially with a random delay in between.

The system is initialized by a 512 KiB data object with random data being written to the quorum system before the experiments starts. Each client repeats the operation 50 times, and the fastest and slowest operations are removed in order to mitigate the effect of outliers. The final operation latency result is the average of every client's average operation latency. Taking the average over all clients accounts for local variations, since different PlanetLab nodes have different conditions.

PlanetLab servers do not have any uptime guarantees, and we therefore want to allow a few servers to fail (i.e., $f > 0$). But because k is bounded to be an integer value such that $1 \geq k \geq N - 2f$, f can not be chosen freely. It therefore stands clear that if f is constant, N can never be chosen such that k would be forced to be less than 1. Conversely, since we want to run an experiment with as few as five servers, that puts a bound on f. The table in Appendix B shows all allowed values for k, and demonstrates very succinctly that the maximal f that can be chosen for five servers is 2. Because of this, f was chosen to be fixed to 2 for the experiments.

6.2.2 Client Scalability Experiment

This scenario is made to evaluate how the read and write latency are affected when increasing the number of writers and readers respectively. This tests the ability of the servers to handle an increase of concurrent client operations. The number of allowed node failures is kept constant, which means that the quorum size is also constant. Both the read and write operation latency is measured. The settings are summarized in Table A.1 and Table A.2.

6.2.3 Server Scalability Experiment

The server scalability experiment is constructed to evaluate in what way the read and write latency are affected when increasing the number of servers. The number of allowed node failures is kept constant, which means that the quorum size grows with the number of servers. So when the servers increase, the number of servers that a client has to access will also increase but the coded elements will be smaller in size. One interesting aspect to look at when increasing the number of servers is whether the effect of higher code rate trumps the effects of having a larger quorum. Both the read and write operation latency is measured. The settings are summarized in Table A.3.

6.2.4 Data Object Scalability Experiment

To evaluate how the read and write latency are affected by the size of the stored data object, this experiment performs operations using increasingly large data objects. The size is increased to a maximum of 4 MiB, which was found to be enough to see the scalability. These algorithms are not particularly suitable for replicating very large data objects, and if that is expected to be the common, algorithms which are optimized for that use case (like [11]) would be a better choice.

The number of allowed node failures is kept constant, as well as the number of servers, which means that the quorum size is also constant. The experiment is run in isolation from other client nodes, so that scalability in increasing data object sizes can be reliably measured. Both the read and write operation latency are measured. The settings are summarized in Table A.4.

6.2.5 Reset Experiment

This scenario measures how long it takes for the servers to reset their local state after a transient fault. Since this part requires that all servers participate, we do not allow any servers to be unresponsive. Because some nodes on PlanetLab were highly unstable, it was hard to run experiments for prolonged stretches of time. Therefore, we limited the number of repetitions for the reset experiment (which was expected to take longer than the other experiments) to 20 instead of 50.

Having to reset the global system state is the worst case scenario when it comes to convergence after a transient fault. The time measured is from a client pre-write phase (with a maximal tag number) until a query operation is successful. In order to know that every server has finished the reset phase, f is set to 0, meaning the client has to receive responses from all servers before returning. The settings are summarized in Table A.5.

6.2.6 Overhead Experiment

In this scenario, we compare the overhead of the self-stabilizing version of CAS to the original CAS algorithm. To evaluate this, we have two implementations of the original CAS. The first is a modified version of the self-stabilizing variant, where parts of the implementation related to self-stabilizing mechanisms has been removed. The second experiment is a more performance optimized version using ZeroMQ sockets. More about the implementations can be found in Section 5.4. The settings are summarized in Table A.6.

7

Evaluation Results

In this chapter, we present and discuss the results of each of the evaluation experiments. We start by looking at the two client scalability experiments, and then at the server scalability experiment. Those are followed by the data object scalability experiment and reset time experiment. Last but not least we have the two overhead experiments, where we compare the overhead of the self-stabilizing version of CAS (called CASSS) with two different implementations of the traditional CAS algorithm.

Our results show that the self-stabilizing version of CAS that we have implemented is efficient and scalable. Compared to the traditional CAS algorithm, it has only a constant overhead in terms of operation latency. It is efficient in storing up to 1 MiB of data, and can perform a global reset within a few seconds for systems with up to 20 servers.

7.1 Client Scalability

In this section, the results of the client scalability experiments are presented and discussed. Figure 7.1 shows the result of the experiment where the number of concurrent readers was changed, and Figure 7.2 the corresponding experiment for number of concurrent writers. Both graphs show a pretty flat curve, which indicates that none of the experiments reached a point where the system was overwhelmed by the number of concurrent operations.

A more interesting point is to understand what causes the difference between each of the operations. The fact that the ABD read operation is the slowest of the four is not much of a surprise. Not only does ABD send larger messages because of the

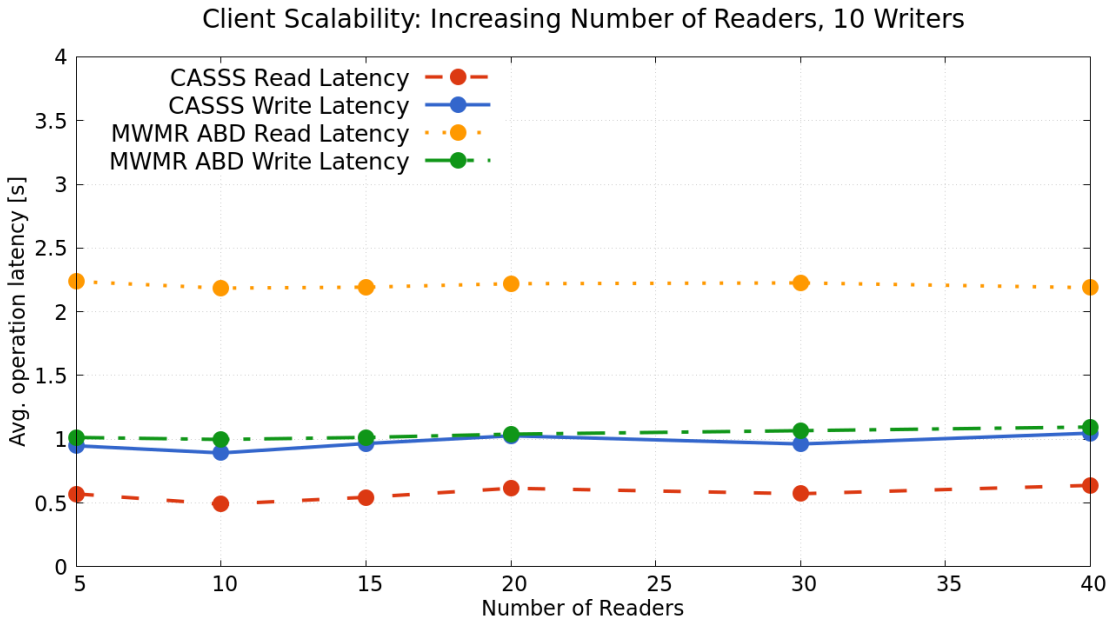


Figure 7.1: Operation latency with respect to the number of concurrent readers.

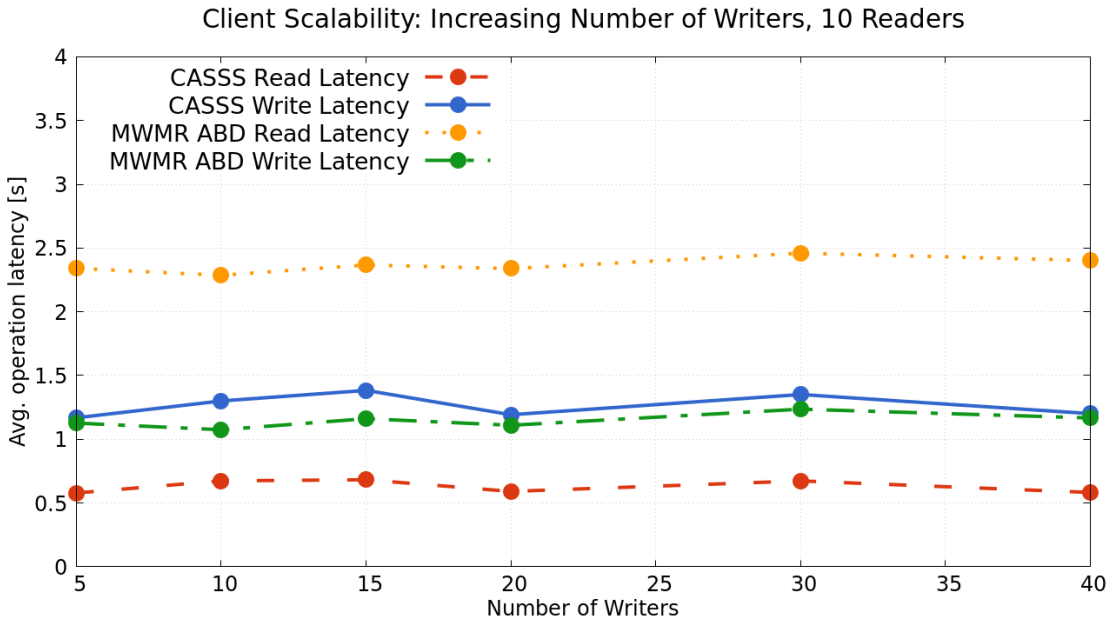


Figure 7.2: Operation latency with respect to the number of concurrent writers.

lack of coding, but the read operation actually transfers data twice: once to fetch the data from the servers, and once during the propagation phase.

The ABD write operation and CASSS write operation completes in about the same amount of time. While the CASSS write operation has two more rounds of communication to perform than ABD write, ABD messages are larger due to the lack of coding. Considering the relatively short RTT between PlanetLab nodes (see Table 7.1), the cost of two extra rounds seems to be about as expensive as the cost of larger messages.

Hostname	**min**	**max**	**avg**
pl1.uni-rostock.de	27.50	79.59	45.59
pl2.uni-rostock.de	27.47	79.57	44.87
planet4.cs.huji.ac.il	65.84	131.41	84.20
planetlab11.net.in.tum.de	15.76	67.85	32.61
planetlab13.net.in.tum.de	15.67	68.68	33.31

Table 7.1: Table of the average ping time (in milliseconds) from each of the five client nodes to all server nodes on our slice on PlanetLab.

Last but not least, we find that the CASSS read operations are the fastest ones. This too was expected, since it has as few rounds of communication as ABD write, but uses coding which decreases the message size.

7.2 Server Scalability

The result of the servers scalability experiment can be found in Figure 7.3. The first observation to make is that with five servers, both the read and write operations for the self-stabilizing CAS, as well as the ABD write operation, ends up at more or less the same spot. That is because with only five servers, CASSS effectively performs full replication and the CASSS quorum size is equal to majority quorum. While ABD read has fewer rounds than CASSS write, ABD read transfers more data which is why it is the slowest of the operations.

Looking at the interval between five and ten servers, the operation latency of ABD increases while the operation latency of CASSS decreases or stays the same. That is because when increasing the number of servers, the quorum size grows but so does the code rate. So while both ABD and CASSS waits for responses from more servers, CASSS gains the advantage of decreased message size.

The coding library we use has a limitation that $k + m \leq 32$. Because of this, f could not be kept at 2 for quorum systems with 20 and 30 servers. For 20 servers, f had to be at least 4, and for 30 servers it had to go all the way up to 14. The point where f is changed is marked by the dashed vertical line in the graph.

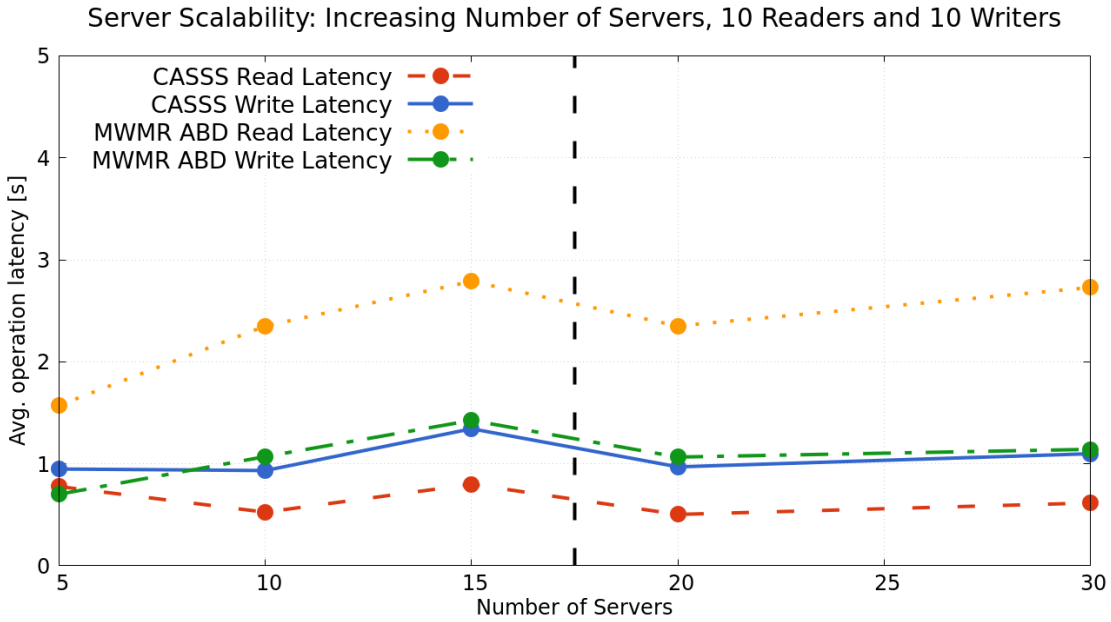


Figure 7.3: Operation latency with respect to the number of servers. The dashed vertical line denotes the point where the parameter f had to be changed.

7.3 Data Object Scalability

This section looks at the results from the data object scalability experiment, shown in Figure 7.4. Other algorithms have previously been shown to be more suitable than CAS and ABD for large data objects, for example [11]. Up until about 1 MiB, the operation latency is fairly minimal. ABD begins to escalate already at 512 KiB, but CASSS is reasonably fast all the way to 4 MiB. This is of course a consequence of the coding, which effectively reduces the message size.

It is also worth pointing out that the multiplexing chunk size (discussed in Section 5.2.4) was set to 1024 KiB during these experiments, which might contribute to the degrading performance after that point (on top of the algorithms being unsuitable for very large data objects).

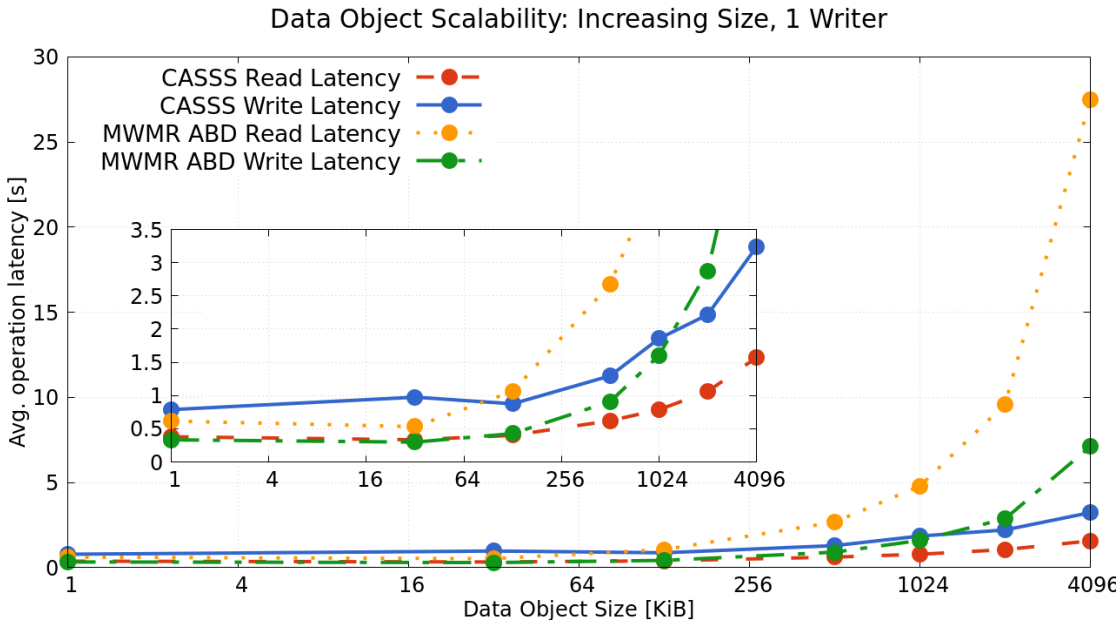


Figure 7.4: Operation latency with respect to the size of the data object.

7.4 Global Reset

The mechanism for global reset should only be triggered because of a transient fault, and so should only have to be run very rarely. Even so, it is still important that it the reset time has a realistic bound. And as we can see in Figure 7.5, up to 20 servers it takes almost as short time as two client write operations, which is rather efficient for such a rare occurrence. Comparing with the other experiments, where the operation latency is in the order of one second, it is not bad at all to see a reset time of just a few seconds.

With more servers, the trade-off of having to contact all N servers starts to show. As the number of servers increase, the likelihood of having to wait for one really slow server rapidly increases too. If the responsiveness for a server at a given time is normally distributed, the likelihood of having one or more slow servers in the system increases exponentially.

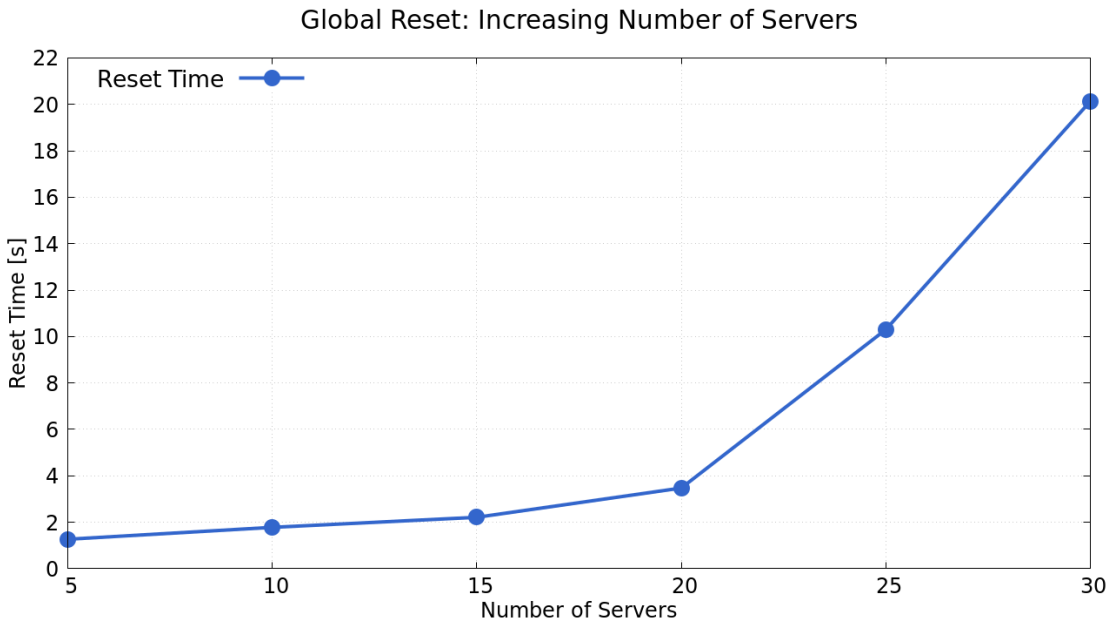


Figure 7.5: The time it takes for the Global Reset mechanism to complete, with respect to the number of servers.

7.5 Overhead

In this section, we present and discuss the results of the experiments where the self-stabilizing version of CAS is compared to implementations of the traditional CAS algorithm. The two experiments demonstrate two different perspectives: Figure 7.6 reveals the overhead that the extra communication round and intensive gossiping have, while Figure 7.7 additionally shows the overhead caused by our implementation of the self-stabilizing token-passing channel. Both figures have a vertical dashed line, which indicates at which point the variable f was changed due to the coding library requirement discussed previously.

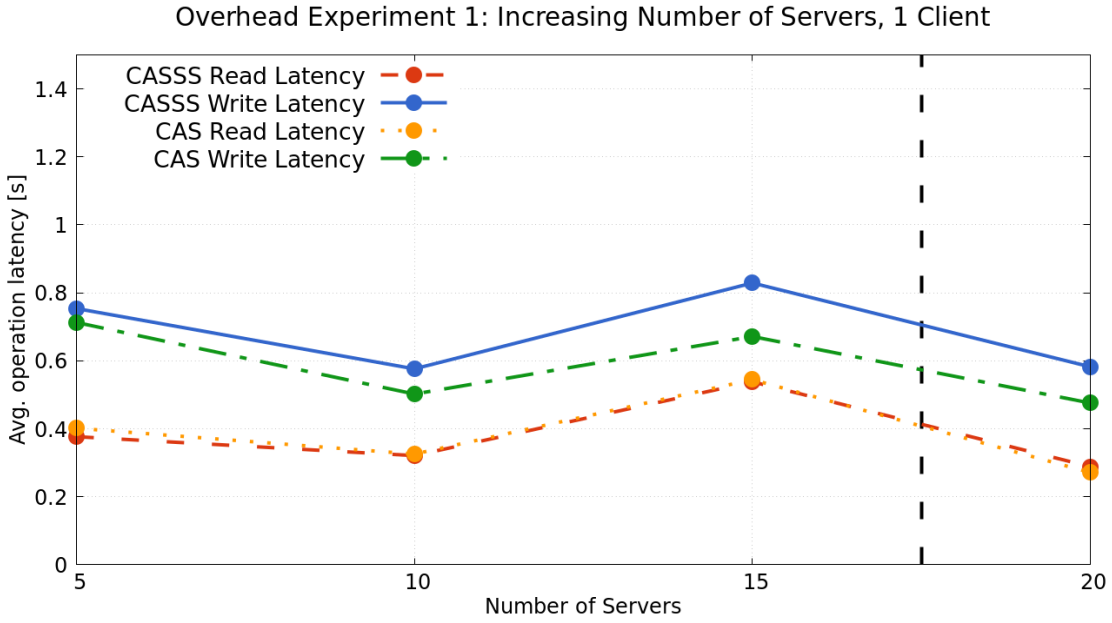


Figure 7.6: Comparison between the operation latency of CASSS versus the traditional CAS algorithm. The dashed vertical line denotes the point where the parameter f had to be changed.

First of all we note that CASSS read and CAS read are nearly identical. This is exactly what one would expect, since the self-stabilizing version of CAS has the same number of rounds for the read operation as the traditional CAS algorithm. The write operations differs slightly, and with the self-stabilizing version needing one extra communication round to complete the write operation, we expected it to be slightly slower than the traditional CAS. The ping time between the PlanetLab nodes were about 50 ms, so the expected cost for one round of communication is consistent with what we find in Figure 7.6.

In the second overhead experiment, we compare our implementation of CASSS with an implementation of CAS which is using ZeroMQ for communication instead of the self-stabilizing channels. In Figure 7.7 it can be seen that each plot follow the same structure. This is expected since they almost run the same algorithm and any difference should be added as a constant factor. The reason that the latency decreases when scaling up the servers from 5 to 10 is the reduced size of the coded elements. When going from 15 to 20 servers, we once more find that there is a decrease in latency. This is due to the aforementioned limitation of the erasure coding library, which forced us to change parameters to reduce the quorum size.

Looking at the difference between CAS ZeroMQ read and write in Figure 7.7, we see that their difference in operation latency is minor. The reason for the difference is the additional communication round needed for the write operation. The CASSS implementation is overall slower, which is expected because of the communication overhead in the self-stabilizing communication channel. Even though it is slower, the overhead is *constant* which is desirable for good scalability.

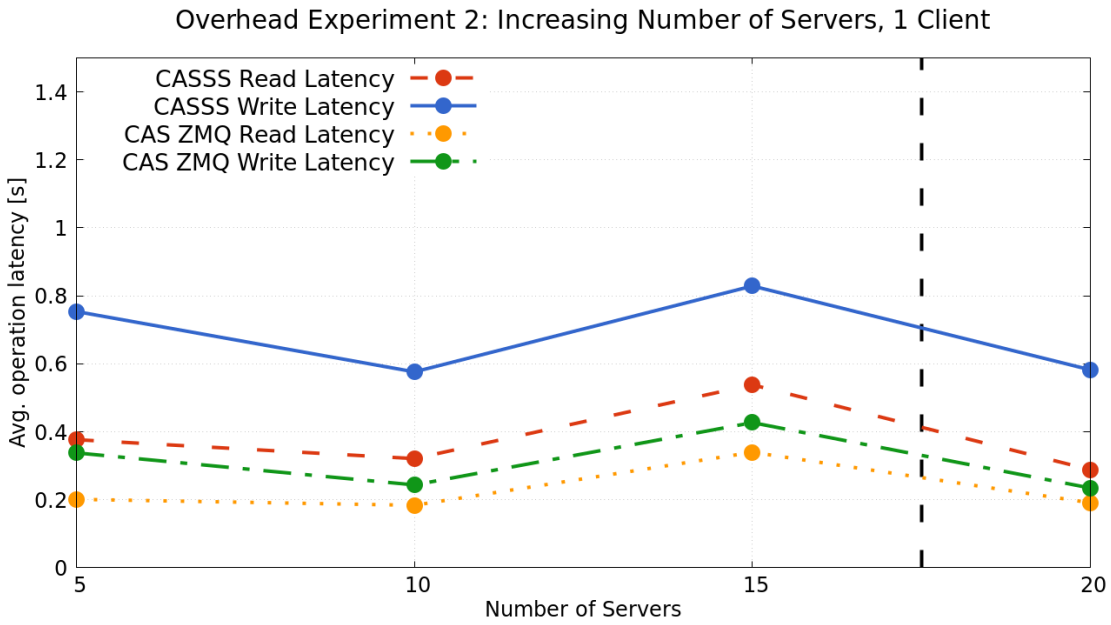


Figure 7.7: Comparison between the operation time of an implementation of CAS and the self-stabilizing version of CAS. The dashed vertical line denotes the point where the coding had to be changed.

8

Discussion

In this chapter, we first compare our work to similar projects from the literature. Then we proceed to discuss what can be done in future work, and end with a conclusion of the project.

8.1 Comparison with Literature

In this section we compare our work with two other implementations and evaluations of similar algorithms, both of which used PlanetLab as evaluation platform.

Vacana [8] did an implementation and evaluation of a self-stabilizing version of a SWMR ABD. The implementation uses epochs to overcome transient faults, compared to our implementation that does a global reset of the sequence numbers instead. Their implementation relies entirely on TCP as transport protocol and we make use of a self-stabilizing communication channel that also overcome communication deadlock. Their implementation of the self-stabilizing ABD has shorter operation latency than our implementation of the Self-Stabilizing CAS. This is most likely because of a combination of chosen programming language, replicating smaller data objects (up to 320 bits), being a SWMR algorithm and not using a self-stabilizing communication channel.

Nicolaou and Georgiou [10] did an experimental evaluation of four MWMR register emulation algorithms and one of them was the *SIMPLE* algorithm. Their algorithm implementation is programmed in C++ and use TCP as transport protocol. Their implementation of SIMPLE achieves lower operation latency than our implementation of CASSS, but is not self-stabilizing. In most of our experiments, we use 512 KiB large data objects, for the benefits of erasure coding.

8.2 Extensions

We believe that the same principles that we use in this project can be applied to other algorithms, in order to make them self-stabilizing. This could for example be applied to other algorithms for distributed storage, like LDR by Fan and Lynch [11]. It could also, we believe, be used to create a self-stabilizing consensus protocol based on Paxos [30]. This could lead to new areas of application for those algorithms, with the stronger fault tolerance guarantee given by self-stabilization.

An extension to replace the TCP file transfer with a TCP friendly rate control for UDP would be a worthwhile effort to optimize the communication. The modular structure of our software makes it easy to change the underlying functionality of the channel without any major changes to the rest of the software. Resources for such an extensions are for example the DCCP protocol [31] and the various RFC for congestion control for UDP [32, 33, 34, 35, 36, 37].

Another extension would be to add a reconfiguration mechanism, in order to allow servers to dynamically leave and join the quorum system. CAS allows for up to f simultaneous server failures, so if there are more crashed servers than that, they need to be removed from the quorum system. Without a mechanism to add functioning servers to the quorum system, the system might eventually reach a quorum collapse. There are algorithms for quorum reconfiguration, for example [1] and [13], that allows for servers to leave and join the quorum system.

8.3 Conclusion

In this report we show how to create self-stabilizing applications for emulating shared memory. It has been shown to work on a real network, the PlanetLab EU platform. There is a small overhead caused by the extension which makes CAS self-stabilizing, and this (albeit slight) trade-off between latency and robustness should be taken into account when considering to implement the algorithm. Some modules of the system may be useful independently for other systems, like for example the reset mechanism which offers synchronized reset of sequence numbers.

Bibliography

[1] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. "Self-stabilizing Reconfiguration". In: *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*. 2017, pp. 51–68. DOI: 10.1007/978-3-319-59647-1_5.

[2] OpenFog Consortium Architecture Working Group. "OpenFog Reference Architecture for Fog Computing". In: *OPFRA001* 20817 (2017), p. 162.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing memory robustly in message-passing systems". In: *Journal of the ACM* 42.1 (1995), pp. 124–142. ISSN: 00045411. DOI: 10.1145/200836.200869. URL: http://dl.acm.org/citation.cfm?id=200869.

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

[5] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. "A coded shared atomic memory algorithm for message passing architectures". In: *Distributed Computing* 30.1 (2017), pp. 49–73.

[6] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. "Self-Stabilizing and Private Distributed Shared Atomic Memory in Seldomly Fair Message Passing Networks". In: *CoRR* abs/1806.03498 (2018). arXiv: 1806.03498.

[7] Shlomi Dolev, Swan Dubois, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. "Crash resilient and pseudo-stabilizing atomic registers". In: *International Conference On Principles Of Distributed Systems*. Springer. 2012, pp. 135–150.

[8] Despina Vacana. "Implementation and Experimental Evaluation of a Self-Stabilizing Atomic Read/Write Register Service". MA thesis. University of Cyprus.

[9] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. "Pragmatic Self-stabilization of Atomic Memory in

Message-passing Systems". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 19–31. ISBN: 978-3-642-24549-7. URL: http://dl.acm.org/citation.cfm?id=2050613.2050617.

[10] Nicolas Nicolaou and Chryssis Georgiou. "On the practicality of atomic MWMR register implementations". In: *Proceedings of the 2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012* September 2011 (2012), pp. 340–347. DOI: 10.1109/ISPA.2012.51.

[11] Rui Fan and Nancy A. Lynch. "Efficient Replication of Large Data Objects". In: *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*. Ed. by Faith Ellen Fich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 75–91. ISBN: 3-540-20184-X. DOI: 10.1007/978-3-540-39989-6_6.

[12] Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. "A Layered Architecture for Erasure-Coded Consistent Distributed Storage". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. PODC '17. Washington, DC, USA: ACM, July 2017, pp. 63–72. ISBN: 978-1-4503-4992-5. DOI: 10.1145/3087801.3087832. URL: http://doi.acm.org/10.1145/3087801.3087832.

[13] Viveck R. Cadambe, Nicolas C. Nicolaou, Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. "ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage". In: *CoRR* abs/1805.03727 (2018). arXiv: 1805.03727. URL: http://arxiv.org/abs/1805.03727.

[14] Nancy A. Lynch and Alexander A. Shvartsman. "RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks". In: *Proceedings of the 16th International Conference on Distributed Computing*. DISC '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 173–190. ISBN: 3-540-00073-9. URL: http://dl.acm.org/citation.cfm?id=645959.676144.

[15] Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. "RADON: Repairable Atomic Data Object in Networks". In: *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Ed. by Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone. Vol. 70. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 28:1–28:17. ISBN: 978-3-95977-031-6. DOI: 10.4230/LIPIcs.OPODIS.2016.28. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7097.

[16] Albin Casparsson and David Gardtman. "A Real-Time Testbed for Distributed Algorithms:Evaluation of Average Consensus inSimulated Vehicular Ad Hoc Networks". 64. MA thesis. 2017.

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

[18] Chryssis Georgiou and Nicolas C Nicolaou. *Simulating efficient mwmr atomic register implementations on the ns2 network simulator*. Tech. rep. Tech. Rep. TR-11-06, Dept. of Computer Science, University of Cyprus, Cyprus, 2011.

[19] E. Brewer. "CAP twelve years later: How the 'rules' have changed". In: *Computer* 45.2 (2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[20] Rich Miller. *Report: Google Uses About 900,000 Servers*. Aug. 2011. URL: http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers (visited on 05/10/2018).

[21] Jack Clark. *5 Numbers That Illustrate the Mind-Bending Size of Amazon's Cloud*. Nov. 2014. URL: https://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html (visited on 05/10/2018).

[22] Lee Mathews. *Just how big is Amazon's AWS business? (hint: it's absolutely massive)*. Nov. 2014. URL: https://www.geek.com/chips/just-how-big-is-amazons-aws-business-hint-its-absolutely-massive-1610221/ (visited on 05/10/2018).

[23] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. "Stabilizing data-link over non-FIFO channels with optimal fault-resilience". In: *Information Processing Letters* 111.18 (2011), pp. 912–920.

[24] 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". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Andréa W. Richa and Christian Scheideler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 133–147. ISBN: 978-3-642-33536-5.

[25] Chryssis Georgiou. "Atomic Registers". In: *Private communications* (2017).

[26] Nancy A. Lynch and Alexander A. Shvartsman. "Communication and Data Sharing for Dynamic Distributed Systems". In: *Future Directions in Distributed Computing, Research and Position Papers*. 2003, pp. 62–67. DOI: 10.1007/3-540-37795-6_13. URL: https://doi.org/10.1007/3-540-37795-6_13.

[27] A. R. Khan, S. M. Bilal, and M. Othman. "A performance comparison of open source network simulators for wireless networks". In: *2012 IEEE International Conference on Control System, Computing and Engineering*. Nov. 2012, pp. 34–38. DOI: 10.1109/ICCSCE.2012.6487111.

[28] *HOWTO Use Linux Containers to set up virtual networks*. URL: https://www.nsnam.org/wiki/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks (visited on 12/12/2017).

[29] Lars Eggert, Gorry Fairhurst, and Greg Shepherd. *UDP Usage Guidelines*. RFC 8085. RFC Editor, Mar. 2017, pp. 1–55. URL: http://www.rfc-editor.org/rfc/rfc8085.txt.

[30] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[31] Eddie Kohler, Mark Handley, and Sally Floyd. "Designing DCCP: Congestion control without reliability". In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. 4. ACM. 2006, pp. 27–38.

[32] Eddie Kohler, Mark Handley, and Sally Floyd. *Problem Statement for the Datagram Congestion Control Protocol (DCCP)*. RFC 4336. RFC Editor, Mar. 2006, pp. 1–22. URL: https://www.rfc-editor.org/rfc/rfc4336.txt.

[33] Eddie Kohler, Mark Handley, and Sally Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340. RFC Editor, Mar. 2006, pp. 1–129. URL: https://www.rfc-editor.org/rfc/rfc4340.txt.

[34] S. Floyd and E. Kohler. *Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control*. RFC 4341. RFC Editor, Mar. 2006, pp. 1–20. URL: https://www.rfc-editor.org/rfc/rfc4341.txt.

[35] S. Floyd, E. Kohler, and J. Padhye. *Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)*. RFC 4342. RFC Editor, Mar. 2006, pp. 1–33. URL: https://www.rfc-editor.org/rfc/rfc4342.txt.

[36] S. Floyd and E. Kohler. *Profile for Datagram Congestion Control Protocol (DCCP) Congestion ID 4: TCP-Friendly Rate Control for Small Packets (TFRC-SP)*. RFC 5622. RFC Editor, Aug. 2009, pp. 1–19. URL: https://www.rfc-editor.org/rfc/rfc5622.txt.

[37] S. Floyd, M. Handley, J. Padhye, and J. Widmer. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 5348. RFC Editor, Sept. 2008, pp. 1–58. URL: https://www.rfc-editor.org/rfc/rfc5348.txt.

A

Experiment Parameters

This appendix lists all the parameter settings used for each of the evaluation scenarios. Coding parameter f is the number of servers allowed to fail. N is the number of servers for each round. R is the number of reader clients and W the number of writer clients.

Nodes		**Parameters**	
Servers (N)	10	f	2
Readers (R)	5, 10, 15, 20, 30, 40	**File Size** (***KiB***)	512
Writers (W)	10	`gossip_freq`	1

Table A.1: Parameters for the reader scalability experiment.

Nodes		**Parameters**	
Servers (N)	10	f	2
Readers (R)	10	**File Size** (***KiB***)	512
Writers (W)	5, 10, 15, 20, 30, 40	`gossip_freq`	1

Table A.2: Parameters for the writer scalability experiment.

Nodes		**Parameters**	
Servers (N)	5, 10, 15, 20, 30	f	2
Readers (R)	10	**File Size** (***KiB***)	512
Writers (W)	10	`gossip_freq`	1

Table A.3: Parameters for the server scalability experiment.

Nodes		Parameters	
Servers (N)	10	f	2
Readers (R)	1	**File Size** (***KiB***)	1, 32, 128, 512, 1024, 2048, 4096
Writers (W)	1	`gossip_freq`	1

Table A.4: Parameters for the data object scalability experiment.

Nodes		Parameters	
Servers (N)	5, 10, 15, 20, 25, 30	f	0
Readers (R)	0	**File Size** (***KiB***)	0.25
Writers (W)	1	`gossip_freq`	1

Table A.5: Parameters for the reset experiment.

Nodes		Parameters	
Servers (N)	10	f	2
Readers (R)	1	**File Size** (***KiB***)	512
Writers (W)	1	`gossip_freq`	1

Table A.6: Parameters for the overhead experiment.

B

Coding Parameter k

This appendix is a cheat sheet for deciding on the coding variable k depending on the number of servers (N) and number of allowed server failures (f). Table B.1 shows the maximum value k can have, while the lowest is always 1. Cells without a value indicate that k would need to be less than 1, which is illegal. More formally, it must hold that $1 \leq k \leq N - 2f$.

N \ f	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	5	3	1								
10	10	8	6	4	2						
15	15	13	11	9	7	5	3	1			
20	20	18	16	14	12	10	8	6	4	2	
30	30	28	26	24	22	20	18	16	14	12	10
40	40	38	36	34	32	30	28	26	24	22	20

Table B.1: The maximal value for the coding parameter k, depending on number the of servers (N) and number of servers allowed to fail (f).

It is worth noting that the coding library `PyECLib` used in project requires $N+k \leq 32$, which must also be taken in consideration when choosing f, N and k.