IMPEMENTATION AND EXPERIMENTAL EVALUATION
OF A SELF – STABILIZING ATOMIC READ / WRITE
REGISTER SERVICE

Despina Vacana

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

June 2014
ABSTRACT

Nowadays, a key problem in distributed computing is the implementation of shared memory in message passing systems. The most fundamental type of a shared data structure to implement is a read/write register, which stores a value. A register is usually replicated among a set of replicas host to provide fault-tolerance and availability. However this distribution introduces the problem of preserving data consistency between the replicas. Atomicity is the strongest consistency guarantee that can be provided in a distributed system. The challenge of providing atomic consistency becomes even greater in the presence of asynchrony and failures either permanent or transient.

The first algorithm that simulates shared memory in memory passing systems is the ABD algorithm [1]. This algorithm simulates a Single-Writer Multi-Reader (SWMR) atomic register in an asynchronous message passing system in which at most half of the processors may crash. The past two decades, remarkable amount of work has been done on improving ABD to work under many types of permanent failures [2]. However, only recently [3], a version of ABD was introduced that incorporates mechanisms for handling transient failures. This algorithm is a self-stabilizing version of the classical ABD algorithm that permits automatic recovery of a system from any kind of transient errors.

The aim of this thesis is the implementation and the experimental evaluation of the algorithm of [3]. In particular is attempted to evaluate the practicality of the algorithm by implementing and deploying it on PlanetLab [3]. In addition is attempted to compare it with the classical ABD algorithm so as to investigate the extra overhead that is required for self-stabilization.
IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF
A SELF – STABILIZING ATOMIC READ / WRITE REGISTER
SERVICE

Despina Vacana

A Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
at the
University of Cyprus

Recommended for Acceptance
by the Department of Computer Science
June, 2014
IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF A SELF – STABILIZING ATOMIC READ / WRITE REGISTER SERVICE

Presented by
Despina Vacana

Research Supervisor

__________________________________
Dr. Chryssis Georgiou

Committee Member

__________________________________
Dr. Vasos Vassiliou

Committee Member

__________________________________
Dr. Nicolas Nicolaou

University of Cyprus

June, 2014
CREDITS

First of all I warmly thank my research supervisor Dr. Chryssis Georgiou for trusting me and giving me the opportunity to work on this topic as well as for the continuing support that provided me during the preparation of my thesis. He was always willing to assist me and encourage me anytime I needed his help.

Furthermore I would like to thank my fellow students Christina Georgiou, Natalie Temene and Marita Thoma, my team members in the postgraduate course "EPL 671 - Research Methodologies in Computer Science" for long and meaningful discussions we had throughout the semester. They helped me to better understand notion of self-stabilization and its importance in distributed computing.

Moreover I would also like to thank my family and my friends for bearing with me, believing in me and supporting me all through these years. Especially, I am heartily thankful to my brother and roommate Agathoklis for brainstorming with me and providing assistance whenever he could.

Finally I thank all those who trusting me and I hope one day to dwell worthy of their expectations.
# TABLE OF CONTENTS

Chapter 1: Introduction ................................................................................................................. 1

1.1 Motivation and Related Work......................................................................................... 1

1.2 Contribution .................................................................................................................... 3

1.3 Thesis Organization ......................................................................................................... 4

Chapter 2: Background Knowledge .............................................................................................. 5

2.1 Self-Stabilization ............................................................................................................. 5

2.2 Asynchronous Message Passing Systems ........................................................................ 8

2.3 Atomic R/W Registers .................................................................................................. 10

2.4 Quorum Systems ........................................................................................................... 12

2.5 TCP/IP Sockets .............................................................................................................. 13

Chapter 3: Description of the Algorithms .................................................................................. 16

3.1 ABD Algorithm Description .......................................................................................... 16

3.1.1 Algorithm’s System Model ....................................................................................... 16

3.1.2 Writer’s Operation ..................................................................................................... 18

3.1.3 Reader’s Operation ..................................................................................................... 19

3.2 Problems of ABD Algorithm under Transient Failures ................................................ 22

3.3 Self-Stabilizing version of algorithm ABD (SelfS-ABD) .................................................. 24

3.3.1 SelfS-ABD Algorithm’s System Model ................................................................. 25

3.3.2 Epochs: A way to overcome transient faults .......................................................... 26

3.3.3 Bounded Labelling Scheme with Uninitialized Values ............................................ 26

3.3.4 Main Variables .......................................................................................................... 29

3.3.5 Algorithm’s Operations ............................................................................................ 30

3.3.6 Repeatedly Information Diffusion .......................................................................... 38
Chapter 4: Implementation of the Algorithms ................................................................. 40

4.1 Establishing System Communication Network ................................................. 40
4.2 Executing Operations ......................................................................................... 42
4.3 Terminating Execution ....................................................................................... 45
4.4 ABD Communication Protocol ......................................................................... 45
4.5 SelfS-ABD Communication Protocol .............................................................. 46

Chapter 5: Experimental Evaluation ........................................................................ 47

5.1. The PlanetLab Platform ................................................................................... 47
5.2. Experimental Platform .................................................................................... 49
5.3. Experimental Scenarios .................................................................................. 51
5.4. Experimental Results ...................................................................................... 52

Chapter 6: Conclusion .............................................................................................. 68

6.1 Conclusions ....................................................................................................... 68
6.2 Future Work ....................................................................................................... 70

References .................................................................................................................. 71
# LIST OF TABLES

Table 1: ABD Communication Protocol Operations .................................................. 46
Table 2: SelfS-ABD Communication Protocol Operations ........................................ 46
Table 3: List of machines that hosted the experiments .............................................. 49
Table 4: SelfS-ABD Experimental Parameters .......................................................... 51
Table 5: Scenario 1 Results - Average writer operation latency in msec .................... 53
Table 6: Scenario 1 Results - Average readers operation latency in msec .................. 54
Table 7: Scenario 1 Results – Maximum size of the messages in bits .......................... 57
Table 8: Scenario 2 Results ....................................................................................... 59
Table 9: Scenario 3 Results ....................................................................................... 62
Table 10: Scenario 3 Results – Convergence Time .................................................... 64
Table 11: Scenario 3 Results - Average writer operation latency in msec .................. 65
Table 12: Scenario 3 Results - Average readers operation latency in msec ............... 66
LIST OF GRAPHS

Graph 1: Scenario 1 - Writer Operation Latency ................................................................. 53
Graph 2: Scenario 1 - Readers Operation Latency ............................................................. 55
Graph 3: Scenario 1 – Maximum size of Messages ............................................................ 57
Graph 4: Scenario 2 - SelfS-ABD Behavior with 9 Readers ............................................. 59
Graph 5: Scenario 2 - SelfS-ABD Behavior with 19 Readers ........................................... 60
Graph 6: Scenario 2 - SelfS-ABD Behavior with 29 Readers ........................................... 60
Graph 7: Scenario 2 Vs Scenario 3 ...................................................................................... 63
Graph 8: Scenario 3 Results – Convergence Time ............................................................ 64
Graph 9: Scenario 3 - Writer Operation Latency ................................................................. 65
Graph 10: Scenario 1 - Readers Operation Latency ........................................................... 66
LIST OF FIGURES

Figure 1: Self-Stabilization - States of the System [Stéphane Devismes (CNRS, LRI)]........ 6
Figure 2: Typical Message-Passing System Configuration ............................................. 8
Figure 3: FIFO Communication Link ........................................................................ 9
Figure 4: Atomicity Example.................................................................................... 12
Figure 5: Atomicity Violation Example................................................................. 12
Figure 6: Majority Quorum System Example......................................................... 13
Figure 7: TCP/IP Socket Session Events ............................................................. 15
Figure 8: Initial ABD Network Configuration Example........................................ 17
Figure 9: ABD Write Operation............................................................................ 19
Figure 10: Atomicity of reads Violation Example [16]........................................ 20
Figure 11: ABD Read Operation Round 1............................................................ 21
Figure 12: ABD Read Operation Round 2............................................................ 21
Figure 13: Distributed Reset – Specification Violation Example.......................... 24
Figure 14: Pseudocode for function Nextb [3].................................................... 28
Figure 15: Pseudocode for function Nexte (S is the set of epoch labels appearing in S) [3] .... 29
Figure 16: QuorumWrite Operation Basic Steps.................................................... 31
Figure 17: Pseudocode for receiving a QuorumWrite message by writer [3]........... 32
Figure 18: Pseudocode for receiving a QuorumWrite message by a reader [3]........... 32
Figure 19: QuorumRead Operation Basic Steps.................................................... 33
Figure 20: Pseudocode of SelfS-ABD Write operation [3]....................................... 34
Figure 21: SelfS-ABD Write Operation Basic Step – Round 1................................. 35
Figure 22: SelfS-ABD Write Operation Basic Step – Round 2................................. 35
Figure 23: Pseudocode of SelfS-ABD Read operation [3]....................................... 36
Figure 24: SelfS-ABD Read Operation Basic Steps – Round 1................................. 37
Figure 25: SelfS-ABD Read Operation Basic Steps – Round 2 ........................................ 37
Figure 26: Pseudocode for receiving a Gossip message by writer [3]............................... 38
Figure 27: Pseudocode for receiving a Gossip message by reader [3]............................... 39
Figure 28: Establish Communication Network .................................................................. 41
Figure 29: Pseudocode for Establishing Communication Network ................................. 42
Figure 30: alarm(wInt) Example ....................................................................................... 43
Figure 31: select() Pseudo Code ....................................................................................... 44
Figure 32: PlanetLab Europe [21] .................................................................................... 48
# LIST OF ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>Attiya, Bar-Noy and Dolev Algorithm</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>MWMR</td>
<td>Multiple-Write Multiple-Read Register</td>
</tr>
<tr>
<td>R/W</td>
<td>Read / Write</td>
</tr>
<tr>
<td>SelfS-ABD</td>
<td>Self - Stabilizing ABD Algorithm</td>
</tr>
<tr>
<td>SWMR</td>
<td>Single-Write Multiple-Read Register</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter motivates the work of the master thesis and outlines its contribution.

1.1 Motivation and Related Work

A distributed system is a collection of independent processes that appears to its users as a single coherent system [4]. Communication in this kind of systems is always based on low-level message passing as offered by the underlying network, namely processors communicate by sending messages over communication links.

However, designing algorithms in a message passing system is harder than in a shared memory system, where processes communicate by writing and reading to shared registers. Thus many distributed algorithms emulate shared memory.

The implementation of shared memory in message passing systems is archived using distributed objects called registers [5]. These objects are distributed in different locations and their consistency is ensured through atomic access despite the present of failures.
The first algorithm that emulates shared memory in memory passing systems is the one by Attiya, Bar-Noy and Dolev known as algorithm ABD [1]. Algorithm ABD allows us to emulate a Single-Writer Multi-Reader (SWMR) atomic register in an asynchronous message passing system in which at most half of the processors may crash.

Distributed systems are vulnerable to failures, which could be either permanent or transient. Therefore, it is important distributed algorithms such as ABD to deal with both permanent and transient failures. As described in [2] the past two decades remarkable work has been done on improving ABD to work under many types of permanent failures. However none of these ABD versions incorporate mechanisms for handling transient failures. Such failures may cause, for example, from incorrect system initialization or from a corrupted message.

In October 2011 at the 13th International Symposium on Stabilization, Safety and Security of Distributed Systems (SSS) N. Alon et al. presented a fault-tolerant and stabilizing emulation of a SWMR atomic register in an asynchronous message-passing system where a minority of processors may crash [3]. This emulation is a self-stabilizing [6] version of ABD which permits automatic recovery of a system from any kind of transient error. To the best of our knowledge this algorithm is the first version of ABD which handles transient failures. In fact it is the first algorithm emulating R/W registers in message passing systems that handles transient failures. In the sequel is used the term SelfS-ABD for referring to N. Alon et al. algorithm.

Self – Stabilization is a concept of fault tolerance in distributed computing. A distributed system is said to be self-stabilizing if no matter the state it is initialized it will reach a desire state.

Despite the fact that SelfS-ABD algorithm was presented just three years, its practicality and effectiveness over realistic, planetary-scale adverse environments has not been studied yet.
The objective of this thesis is to evaluate the practicality of algorithm SelfS-ABD, by implementing and deploying it on PlanetLab [7], a planetary-scale network platform.

1.2 Contribution

The contribution of the thesis is the implementation and experimental evaluation of a Self-Stabilizing Atomic Read / Write Register. In particular the major goal of this thesis is the implementation and the experimental evaluation of algorithm SelfS-ABD [3].

Our research started with the study of papers and previous works related to our topic. This helped us to understand all the information that were essential for the implementation of algorithm SelfS-ABD. After that, we implemented the classical algorithm ABD [1] using C programming language. Afterwards we implemented algorithm SelfS-ABD [3]. Finally we tested our algorithms on PlanetLab.

Through the experimental evaluation, we have investigated the self-stabilization nature of algorithm SelfS-ABD. Also we have identified the overhead required to do so when compared with the classical ABD algorithm. A general conclusion from the experimental evaluation is that the time and space complexity of algorithm SelfS-ABD are larger than that of classical ABD.

To the best of our knowledge our work is the first attempt to study the practicality of algorithm SelfS-ABD and to empirical evaluate its behavior.
1.3 Thesis Organization

In Chapter 2 we present the necessary background knowledge for this thesis. Afterwards in Chapter 3 we give an extended description of ABD and SelfS-ABD. Also in this chapter we point out the problems of ABD when exist transient errors. After that, in Chapter 4 we describe the key points of the implementation of those algorithms. Then in Chapter 5 we provide the experimental evaluation and the results obtained from the experiments. Finally in Chapter 6 we summarize the major conclusions obtained through the experimental evaluation of both algorithms.
Chapter 2

Background Knowledge

This chapter provides necessary background information. Specifically the following concepts are explained: self-stabilization, message-passing systems, atomic R/W registers, quorum systems and TCP/IP sockets.

2.1 Self-Stabilization

Distributed systems are vulnerable to failures, which could be either permanent or transient. Self-stabilization is the ability to automatically resume with constant time normal operation following transient failures [8] which may cause, for example, from incorrect system initialization or from a corrupted message. Also, according to [6] self-stabilization is a powerful technique that permits automatic recovery of a system from any kind of these errors.

Self-stabilization firstly introduced by Edsger W. Dijkstra in 1974. Specifically in [6] Dijkstra presented the first self-stabilizing algorithm for the mutual exclusion problem in order to explain the concept of self-stabilization. However the scientific community began to show more interest to this concept after ten years, when Leslie Lamport in [9] stressed the importance of Dijkstra’s work. Since then self-stabilization remains important as it presents a significant foundation for self-managing computer systems and fault-tolerant systems. It is
worth to be pointed that because of its enormous importance, Dijkstra's paper [6] received the "2002 ACM PODC Influential-Paper Award" which is one of the highest recognitions in the distributed computing community [10]. To commemorate Dijkstra and his significant work after his death this award was renamed to "Dijkstra Award".

The spirit of self-stabilization is to cope with transient faults by designing an algorithm that ensures establishment of a desire property [8]. If a system is self-stabilizing, then it can start from an arbitrary state and it eventually exhibits its desired behavior after a finite number of steps without explicit human intervention. Also whenever occurs an unexpected fault the system can recover automatically by itself within finite time, and will behave correctly afterwards.

Therefore, a system $S$ is self-stabilizing with respect to predicate $P$ that identifies the legitimate states, if it satisfies the following properties:

- **Convergence**: Starting from any arbitrary configuration, $S$ is guaranteed to reach a configuration satisfying $P$, within a finite number of state transitions.

- **Closure**: $P$ is closed under the execution of $S$. That is, once in a legitimate state, it will stay in a legitimate state.

![Figure 1: Self-Stabilization - States of the System (Stéphane Devismes (CNRS, LRI))]
Based on everything mentioned above this characteristic is very powerful and significant. For example, the control system of an autonomous vehicle (e.g. space shuttle, autonomous car) must use a self-stabilizing algorithm for its control. Whenever occurs a fault due a momentary problem, a self-stabilizing algorithm will cause the control to recover automatically after finite number of steps and continue on its task. Note that the vehicle may malfunction during the process of recovering its desired behavior.

In [8], Dolev explains the main idea of a self-stabilizing algorithm using "the stabilizing orchestra example". Dolev urges us to imagine that an orchestra is invited to play in an event. Unhappily the event takes place outdoors on a windy evening and the conductor is absent. The organizers of the event require the orchestra to play the same piece of music all the time. Thus each player has to follow a specific score and whenever he or she finishes playing the last page must be starting again from the beginning. However, during the performance, the wind can turn a few pages of the score, without the players to notice the change. As a result of this the players don't play anymore in harmony. According to Dolev’s self-stabilization solution to this problem every player should join the neighboring player who is playing the earlier page, if this page is earlier than his or her own page. So the player who is playing the page with the minimal page number will cause, eventually, all the other players to join him: first his neighbors will join him and once these neighbors play simultaneously, the neighbors of these will change page and join them and so on. Therefore, finally all players will be playing in harmony. Thus after a transient fault, due a wind’s page turn, the players may be playing a different page for a while, but eventually they will be synchronize again, which is the desire property in the orchestra case. So this abstracts the distributed synchronization problem, a fundamental problem in distributed computing.

To sum up, the self-stabilization approach allows a system to cope with transient faults. After the occurrence of a transient error the system can recover atomically within finite time without any direct human intervention. A self-stabilizing algorithm doesn’t need any initialization and
a self-stabilizing system tolerates communication channels unreliability and dynamic topology changes too. However self-stabilization accepts only transient faults and during the stabilization time the system may reach inconsistent states. As a final note, sometimes it is very challenging to design and develop self-stabilizing systems due to their high complexity. Therefore it is a significant development when one is able to develop such a system.

2.2 Asynchronous Message Passing Systems

A message passing system consists of $n$ processors which are located at the nodes of a network and communicate by sending messages over communication links. A typical message passing system appears in Figure 2.

![Figure 2: Typical Message-Passing System Configuration](image)

The processors of a message-passing system are independent, autonomous and each of them has its own memory. Also the speed of them can be different.

Moreover the communication is asynchronous. A communication link is either unidirectional or bidirectional. A unidirectional communication link from processor $p_1$ to a processor $p_2$
transfers messages from $p_1$ to $p_2$. On the other hand, a bidirectional communication link from $p_1$ to $p_2$ transfers messages from $p_1$ to $p_2$ and form $p_2$ to $p_1$.

When a processor $p_1$ sends a message to a processor $p_2$ there is no guarantee that $p_2$ will eventually receive that message. Should be kept in mind that the messages can be lost, they can be delivered in a different order than the one sent or they can be corrupted; furthermore the communication delay may vary.

A FIFO communication link ensures that messages will eventually be delivered to the receiver in the same order they were sent. This can be done using a FIFO queue for every communication direction. For example, consider two processors $p_1$ and $p_2$. A unidirectional communication link between them is modeled by one FIFO queue while a bidirectional communication link is modeled by two FIFO queues, one from $p_1$ to $p_2$ and the other form $p_2$ to $p_1$. The communication FIFO queue $q_{1,2}$ contains all messages sent by a processor $p_1$ to $p_2$ that have not yet been received. Whenever $p_1$ sends a message to a processor $p_2$ the message is added to the tail of the queue (enqueued) and when the $p_2$ receives the message $m$ that is at the head of the queue, this message is removed from the queue (dequeued).

![Figure 3: FIFO Communication Link](image-url)
The underline communication graph is completely connected if and only if every pair of processors is directly connected, namely every pair of processors have a communication link of bounded capacity c.

Lastly failures may occur at any time. For example processors may fail by stopping. Note that it is very difficult in asynchronous systems to distinguish a slow processor from a failed processor.

2.3 Atomic R/W Registers

The most fundamental type of shared data structure to implement is distributed objects called R/W registers. These objects store a value and can be accessed by two operations; a read operation that returns its value and a write operation that changes its value [5].

Depending on the number of processes that can perform these operations, registers can be classified as Single-Write (only one process may perform write operations), Multi-Write (multiple processes may perform write operations), Single-Read (only one process may perform read operations) and Multi-Read (multiple process may perform read operations).

Therefore, in a SWMR register emulation only one process is allowed to perform write operations while multiple processes are allowed to perform read operations. The process that executes write operations is called "writer" and the processes that execute read operations are called "readers". Therefore, in a SWMR register emulation there is only one writer and many readers. In the same way, in a multiple-write multiple-read (MWMR) register emulation there are many writers and many readers.
In message-passing systems read and write operations can perform either sequentially or concurrently. Also, each operation has an invocation point and a response point. If operations are performed sequentially then an operation $\pi_1$ completes before a second operation $\pi_2$ is called, namely operations are performed one at a time. If this is the case we say that the operation $\pi_1$ precedes operation $\pi_2$. Consequently two operations are concurrent if neither of them precedes the other.

As mentioned earlier, R/W registers are used to implement shared memory in message-passing systems where system components are located in different geographical locations. So it’s important to ensure the consistency between them.

Atomicity (linearizability) [11] is the strongest consistency guarantee that can be provided in a distributed system, while emulating concurrent operations. It requires the results of operations to be as if they were executed sequentially even though they were executed concurrently.

An atomic register has to satisfy the following two properties [1]:

- **Regularity**: Every read operation returns either the value written by the most recent preceding write operation (the initial value if there is no such write) or a value written by a write operation that is concurrent with this read operation.

- **Global ordering**: If a read operation $R_1$ reads a value from a write operation $W_1$, and a read operation $R_2$ reads a value from a write operation $W_2$ and $R_1$ precedes $R_2$, then $W_2$ doesn’t precede $W_1$.

For instance suppose an atomic R/W register with initial value 0. Also suppose two concurrently operations; a write(8) and a read() as appears in Figure 4. Note that the star (*) shows when the operation are executed, called serialization point. In the scenario (a) the read() is executed before the write(8) while in the scenario (b) first is executed the write(8) and after the read(). Following the "regularity" property in the first case (a) the read() returns
the initial value of the register and in the second (b) the \textit{read()} returns the value of the concurrent write operation.

![Figure 4: Atomicity Example](image)

In constant, in the following scenario (Figure 5) the atomicity is violated. The second \textit{read()} operation doesn’t return the value written by the last write that precedes it in the order.

![Figure 5: Atomicity Violation Example](image)

### 2.4 Quorum Systems

Consider a system with \( n \) processes. A \textit{quorum system} is a collection of subsets of these \( n \) processes, with the property that each pair of sets have at least one common process, namely each pair of sets have a non-empty intersection \([2]\). Each subset is called \textit{quorum}.

There are many types of quorum systems such as matrix, crumbling walls and majority \([12]\). The most known and simplest to implement quorum system is the majority quorum system \([13]\) where each quorum has at least \( \lceil (n + 1) / 2 \rceil \) processes. In Figure 6 depicts an example
of a majority quorum system. In particular, 10 processes are grouped into 4 quorums where each of them has 6 processes.

![Figure 6: Majority Quorum System Example](image)

Generally speaking, these systems have been used to implement a wide variety of distributed objects and services such as replicated databases, mutual exclusion, R/W storage and group communication.

When it comes to atomic register implementations, the purpose of a quorum system is to maintain consistency with minimum number of nodes. For example, in one R/W atomic register system of 3 processors, writing a value in any two processes (majority) guarantees that any later read by two processors will return a consistent value.

### 2.5 TCP/IP Sockets

Sockets [14] allow communication between two different processes on the same or different machines. Especially if the processes are on different machines, the communication is achieved by sending messages over the underlying network. For this reason these sockets are
called network sockets. The Internet sockets are a specific kind of network sockets which follow the Internet Protocol standards.

Programmers typically use socket libraries rather than coding directly to lower level socket APIs. The most commonly use socket library for Linux based systems is the Berkeley Sockets [15]. This library provides a set of API functions similar to those programmers use for working with files, such as open(), read(), write() and close().

Technically a socket provides a bidirectional communication endpoint for sending and receiving data with another socket. In particular, there are four types of sockets available to the users: stream sockets, datagram sockets, raw sockets and sequenced packet sockets. The first two are most commonly used rather than the last two. Note that processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

A TCP/IP socket is an Internet stream socket. Such a socket uses the TCP for data transmission. Therefore, this socket is a connection-oriented socket and the delivery in a networked environment is guaranteed. Also a TCP/IP communication ensures that messages will eventually be delivered to the receiver in the same order they were sent. For instance, if sent through the TCP/IP socket three items "A, B, C", they will arrive in the same order to the receiver - "A, B, C". Furthermore if delivery is impossible, the sender receives an error indicator.

TCP/IP sockets are commonly used for client and server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.
In a client-server model the TCP/IP socket on the server process waits for requests from a client. To do this, the server first establishes (binds) an address that clients can use to find it. When the address is established, the server waits for clients to request a service. The client-server data exchange takes place when a client connects to the server through a TCP/IP socket. The server performs the client's request and sends the reply back to the client.

The following figure shows the typical flow of events for a TCP/IP socket session:

![Figure 7: TCP/IP Socket Session Events](image-url)
Chapter 3

Description of the Algorithms

This chapter provides an extended description of the algorithms implemented as part of this master thesis. We describe algorithm ABD [1][2] and then we point out the problems of ABD in the present of transient errors. Finally we describe SelfS-ABD, the self-stabilizing version of ABD [3].

3.1 ABD Algorithm Description

The first algorithm that emulates shared memory in memory passing systems is the one by Attiya, Bar-Noy and Dolev, most known as ABD [1]. Algorithm ABD enables us to emulate a SWMR atomic register in an asynchronous message-passing system in which at most half of the processors may crash. A complete formal proof of ABD algorithm correctness can be found in the original paper [1].

3.1.1 Algorithm’s System Model

Consider a system with $n$ processors $P = \{p_0, p_1, ..., p_n\}$ emulating a shared memory using SWMR atomic registers. Therefore, one processor is the writer which can perform write
operations (processor $p_0$) while the rest of the processors are readers and they can perform *read operations*. In Figure 8 depicts an example of such a system.

Also consider that the underline communication graph is completely connected so every pair of processors is direct connected, namely every pair of processors have a communication link of bounded capacity $c$.

![Figure 8: Initial ABD Network Configuration Example](image)

All processors store a copy of the current value of the register. Each value is associated with a sequence number, an unbounded integer number. This sequence number is increased wherever the writer writes a new value. Also in the initial state of the simulation all sequence numbers are set to zero. So each processor holds a pair $\langle seq\_num, v_i \rangle$.

Algorithm ABD is based on reads and writes to a majority of system’s processors. Specifically, it supports two operations for accessing the register: *write* and *read*. Also it provides and an information (*info*) operation which it is performed always as part of the *read* operation.
In addition processors can crash at any time but algorithm ABD operates correctly if only a minority of processors to crash.

Lastly, the original design of ABD algorithm [1] copes with non-FIFO and unreliable communication links. However, for the purposes of this thesis we assume that the communication between the individual components of the system is achieved by exchanging messages over reliable FIFO communication channels.

3.1.2 Writer’s Operation

There is only one writer in the system, processor $p_0$, and it is responsible for updating the register’s value. Whenever it changes its local replica value it performs a write operation to inform the rest of the processors for that change so to maintain consistency between them.

Specifically, in a write operation the writer sends a $write(seq_{num_{new}}, \nu_{new})$ message, with the new value and an incremented sequence number to all processors and waits for a majority of acknowledgements (see Figure 9). In this way, the writer can be sure that a majority of processors store its latest value.

Whenever a processor $p_i$ receives a message $write(seq_{num_{new}}, \nu_{new})$ message, it changes its local replica and adopts the values contained in the receiving message. So it assigns $seq_{num_i} := seq_{num_{new}}$ and $\nu_i := \nu_{new}$. Then it sends an acknowledgement back to the sender (see Figure 9).
3.1.3 Reader’s Operation

As explained earlier, in the system there is only one writer while the remaining processors are readers. A reader processor $p_i$ can perform read operations in order to find out the most recent value in the system, namely the value with the highest sequence number.

In particular, a reader $p_i$ first queries all processors to find out their local register value and waits at least a majority of responses. After that, the reader picks the value $v_{\text{max}}$ with the highest sequence number and then it assigns $v_{\text{max}}$ and $\text{seq\_num}_{\text{max}}$ as its local register value, namely $\text{seq\_num}_i := \text{seq\_num}_{\text{max}}$ and $v_i := v_{\text{max}}$ (see Figure 11).

Whenever a processor $p_j$ receives a read request it returns back to the sender its local register sequence number, $\text{seq\_num}_j$, and value $v_j$. 
To ensure atomicity of reads a reader $p_i$ propagates the value that it picked. Otherwise the atomicity is violated. In the following example (Figure 10) the first read operation (green) returns the value $v_0$ while the second read operation (red) returns the value $v_1$. This violates the atomicity of reads because the second operation (red) starts after the completion of the first and following the "global ordering" property of the atomicity it must be return the same value with the first one, namely $v_1$.

![Figure 10: Atomicity of reads Violation Example [16]](image)

Therefore, to ensure atomicity of reads $p_i$ sends an $info(seq\_num_{\text{max}}, v_{\text{max}})$ message, with the value it picked and the associate sequence number to all processors.

Whenever a processor $p_j$ receives an $info(seq\_num_{\text{max}}, v_{\text{max}})$ message, it checks if its local register value is the most recent value in whole system, namely it checks if $seq\_num_i$ isn’t smaller than the sequence number contained in the receiving message. If it finds out that its value isn’t up to date it changes it with the receiving values. The $info$ operation terminates when the reader $p_i$ receives a majority of acknowledgements. Then it returns the value that it found (see Figure 12).
**Round 1:**
Discover most recent value in whole system

1. `read()`

2. Message receivers send back their local register value

3. Wait
4. Pick the value with the highest sequence number
   `<seq_num_{max}, u_{max}>`
5. Assign `<seq_num_{max}, u_{max}>` as its local register value

**Figure 11: ABD Read Operation Round 1**

**Round 2:**
Propagate the value that it picked

6. `prev(seq_num_{max}, u_{max})`
7. Message receivers update their local register value iff `seq_num_i < seq_num_{max}`
8. Wait `ACK_{1}` from majority
9. Read completes with `<seq_num_{max}, u_{max}>`

**Figure 12: ABD Read Operation Round 2**
3.2 Problems of ABD Algorithm under Transient Failures

A fundamental hypothesis of ABD is that it starts in an initialized state in which all sequence numbers are zero and only the writer can increase them. Moreover, the sequence numbers are positive integer numbers. If it uses a 64-bit representation of the sequence number, then can exist $2^{64}-1 = 18,446,744,073,709,551,615$ different sequence numbers (and home there so many different writes). Therefore, under normal circumstances, a 64-bit sequence number will not wrap around for a number of writes that lasts longer than the life-span of any reasonable system!

However, when transient faults (ex. incorrect system initialization, corrupted message) occur in the system its behavior may be unpredictable. Note that transient faults influence the state variable of processors (for more details see Section 3.3.1). For example, the simulation may begin at an uninitialized state where sequence numbers are not necessary all zero. Moreover a transient fault may cause the sequence numbers to hold maximal values when the emulation starts running and thus will wrap around very quickly. The work in [17] suggests a distributed reset technique to deal with this problem but this may result to violation of algorithm specifications. Also due to the asynchronous communication, a reset operation does not guarantee that all sequences numbers are set back to zero.

For instance, consider the system configuration as depicted in Figure 13 (1). Also assume that Reader 1 wants to perform a read operation. At first it queries all processors to find out their local register value and waits at least a majority of responses. After that, Reader 1 picks $<200, 2014>$ which has the highest sequence number and then it adopts these values. Next step is to inform others about the value found. Suppose that before sending the message, due to a transient error, its $seq_{num_{max}}$ holds the value $2^{64}-1$. Thus the info message will include the new $seq_{num_{max}}$. When the writer receives this message it will change its local register value to $<2^{64}-1, 2014>$, since the info message contains a sequence number larger than its local
sequence number. Afterwards suppose that the Writer wants to change its local register value to 2015. First it tries to increase the sequence number but it finds out that the sequence number has reached its maximum value and so it resets it. However the correct highest sequence number is 200 so the specifications of the algorithms are violated.

Furthermore according to the above example at any time a corrupted information message can result the change of writer’s sequence number whether this message contains a sequence number larger than writer’s current sequence number.

Because of this sequence number may wrap around. Also this emulation violates the specifications of the simulation since only the writer can increase the sequence number.
3.3 Self-Stabilizing version of algorithm ABD (SelfS-ABD)

In October 2011, at the 13th International Symposium on Stabilization, Safety and Security of Distributed Systems (SSS), N. Alon et al. presented a fault-tolerant and stabilizing emulation of a SWMR atomic register in an asynchronous message-passing system where a minority of processors may crash [3].

This algorithm is the self-stabilizing version of algorithm ABD which permits automatic recovery of a system from any kind of transient errors. To the best of our knowledge, this algorithm, called SelfS-ABD, is the first version of algorithm ABD which tolerates transient failures.

A formal proof of algorithm SelfS-ABD’s correctness can be found in the original paper [3].
We proceed with an overview of the algorithm (system and operation).

### 3.3.1 SelfS-ABD Algorithm’s System Model

The basic system model for this algorithm is similar with the system model of the classical ABD algorithm, which was described in Section 3.1.1. So consider again a system with \( n \) processors emulating a shared memory using SWMR atomic registers; processor \( p_0 \) is the writer and the rest of the processors are readers.

All processors store a copy of the current value of the register. Each value is associated with a timestamp from a bounded labeling scheme defined in [3]. So each processor holds a pair \( <\text{timestamp}_i, \nu_i> \). More details about the timestamp format and the bounded labeling scheme are given below (see Sections 3.3.3 and 3.3.4).

A transient fault is a brief malfunction that it can change the state variable of processors [18]. Such a failure affects the system only for a fixed amount of time and often occurs at irregular and unpredictable times. Also transient failures may result from incorrect system initialization or from a corrupter message.

Furthermore, algorithm SelfS-ABD is based on reads and writes to a majority of system’s processors. Specifically, it’s supporting two operations for accessing the register: a read operation called \textit{QuorumRead} and a write operation called \textit{QuorumWrite}. These operations are described in Sections 3.3.5.1 and 3.3.5.2.

The underline communication graph is completely connected and processors may crash at any time. Like algorithm ABD, algorithm SelfS-ABD allows only a minority of processors to crash.
In contrast to the original algorithm ABD, the algorithm SelfS-ABD requires reliable self-stabilizing bounded FIFO communication channels [19]. Also processors can start at an arbitrary state.

3.3.2 Epochs: A way to overcome transient faults

To overcome the problems which may possibly appear due transient faults the authors of [3] suggest splitting the emulation execution into epochs. An epoch is a period during which the sequence numbers have supposing not wrapped around. Whenever a "corrupted" sequence number is found, a new epoch is begun, overriding all previous epochs. This procedure repeats until there aren't any more "corrupted" sequence numbers in the system, and the system stabilizes. After system stabilization the epoch remains the same at least until the sequence number wraps around again or there is another corruption in the system (transient fault).

Epochs are denoted with labels from a bounded domain using a new bounded label scheme introduced in [3]; we describe it next.

3.3.3 Bounded Labelling Scheme with Uninitialized Values

This scheme, as described in [3], needs no initialization so it can be used when communication links and processes are started at an arbitrary state, as the example due to a transient fault. Furthermore, this scheme safeguards that if the writer eventually learns about all the epoch labels in the system, it will generate a new epoch label greater than all of them (and hence avoid the violation of the algorithms specification).
3.3.3.1 Epochs Labels

Based to this scheme an epoch label is a pair \((s, A)\) where \(s\) is called the *Sting* of the label and \(A\) the *Antistings* of the label. The *Sting* is an integer from a fixed set \(\mathcal{X}\) and the *Antistings* is a subset with \(k \in \mathbb{Z}^+\) elements of that set.

In particular, set \(\mathcal{X}\) is a fixed set of integers which is defined during the initialization of the simulation and has size \(\mathcal{K}\) where \(\mathcal{K} = k^2 + 1, k > 1\) so \(\mathcal{X} = \{1, 2, ..., \mathcal{K}\}\). For example consider that \(k = 3\). Hence set \(\mathcal{X} = \{1, 2, ..., 10\}\), since \(\mathcal{K} = k^2 + 1 \Rightarrow \mathcal{K} = 3^2 + 1 \Rightarrow \mathcal{K} = 10\) and \((1, <2, 3, 4>)\) is an epoch label where *Sting* = 1 and *Antistings* = \(<2, 3, 4>\).

In addition this bounded scheme provides the following antisymmetric comparison predicate \(<_b\) among two epoch labels \((s_1, A_1)\) and \((s_2, A_2)\):

\[
(s_1, A_1) <_b (s_2, A_2) \equiv (s_1 \in A_2) \land (s_2 \notin A_1)
\]

For instance, if \(\ell_1 = (3, <2, 5, 7>)\) and \(\ell_2 = (4, <3, 6, 8>)\) then \(\ell_1 <_b \ell_2\) because \(s_1 \in A_2\) and \(s_2 \notin A_1\).

Furthermore the scheme gives a function \(\text{Next}_b\) that returns, given a set of at most \(k\) epoch labels, a new epoch label which is greater than every label in the given set with respect to comparison predicate \(<_b\). Specifically, given a set of \(k\) epoch labels \((s_1, A_1), ..., (s_k, A_k)\), this function returns a new epoch label \((s_{\text{new}}, A_{\text{new}})\) where:

- \(s_{\text{new}}\) is an element of \(\mathcal{X}\) that is not in the union \(A_1 \cup A_2 \cup ... \cup A_k\) As the size of \(\mathcal{X}\) is \(k^2 + 1\) there is always at least one element in \(\mathcal{X}\) that is not in the given union.
- \(A_{\text{new}}\) is a subset of size \(k\) containing all values \((s_1, s_2, ..., s_k)\). The elements of this subset must be unique, so the function \(\text{Next}_b\) adds arbitrary elements from \(\mathcal{X}\) to get a set of exactly \(k\) elements.
As an example, consider a set of three epoch labels: \( \ell_1 = (3, < 4, 5, 7>) \), \( \ell_2 = (4, < 1, 3, 5>) \) and \( \ell_3 = (5, < 2, 3, 4>) \). In this case \( \ell_{\text{new}} = (6, < 3, 4, 5>) \) can be returned as a result of function Nextb.

An abstract algorithm pseudocode for the Nextb function appears in Figure 14.

3.3.3.2 Timestamps

In the present self-stabilization algorithm, each value is associated with a timestamp. A timestamp is a pair \((\ell, i)\) where \(\ell\) is a bounded epoch label and \(i\) is an integer sequence number between 0 and a fixed bound \(r \geq 1\). This represents the wrap around bound (e.g. \(2^{64}-1\)).

Two timestamps, \((\ell_1, i_1)\) and \((\ell_2, i_2)\), can be compared using the comparison operator \(\prec_e\):

\[
(\ell_1, i_1) \prec_e (\ell_2, i_2) \equiv \ell_1 \prec_b \ell_2 \lor (\ell_1 = \ell_2 \land i_1 < i_2)
\]

For instance, if \(t_1 = ((3, < 2, 5, 7>), 1)\) and \(t_2 = ((3, < 2, 5, 7>), 2)\) then \(t_1 \prec_e t_2\) because \(\ell_1 = \ell_2\) and \(i_1 < i_2\). Moreover, if \(t_1 = ((3, < 2, 5, 7>), 2)\) and \(t_2 = ((4, < 3, 6, 8>), 1)\) then \(t_1 \prec_e t_2\) because \(\ell_1 \prec_b \ell_2\).
Moreover, new timestamps can be generated using the function Next\(_e\). Given a set \( S \) of at most \( k \) timestamps \((\ell_1, l_1), (\ell_2, l_2), \ldots, (\ell_k, l_k)\) the function Next\(_e\) returns a new timestamp \((\ell, l)\) which is greater than every timestamp in the given set with respect to comparison predicate \(<_e\). Particularly, this function attempts to find the greatest timestamp \((\ell_o, l_o)\) in \( S \) with sequence number less than \( r \). If such a timestamp \((\ell_o, l_o)\) exists, it increments its sequence number by one, leaving its epoch label unchanged and then returns it as the new generated timestamp. Otherwise, it applies Next\(_b\) to obtain an epoch label greater than all the ones in \( S \) and then returns a new timestamp composed of this epoch label and a zero sequence number. The pseudocode for function Next\(_e\) appears in Figure 15.

For example, consider a set of 3 timestamps, \( t_1 = ((3, \prec 2, 5, 7 \succ), 2) \), \( t_2 = ((3, \prec 2, 5, 7 \succ), 1) \) and \( t_3 = ((3, \prec 2, 5, 7 \succ), 2) \). Also consider that \( r = 4 \). In this case \( t_{new} = ((3, \prec 2, 5, 7 \succ), 3) \) can be returned as a result of function Next\(_e\).

![Figure 15: Pseudocode for function Next\(_e\) (\( \tilde{S} \) is the set of epoch labels appearing in \( S \)) [3]](image)

### 3.3.4 Main Variables

Each processor \( p_i \) implements a register which stores a variable \((\text{MaxTS}_i, v_i)\) where \( \text{MaxTS}_i \) is a pair \(<ml_i, cl_i>\) and \( v_i \) is the value of the register.
The \( ml_i \) of \( MaxTS_i \) is the timestamp related with the most recent write to the variable \( v_i \) and the \( cl_i \) is a canceling timestamp which must be always greater than \( ml_i \) according to the comparison predicate \( <_e \). This canceling timestamp allows the writer to know if exists a timestamp in the system that is greater of its local timestamp \( MaxTS_{0:ml} \) in the \( <_e \) order. The writer is the only node in the system that can increase the value of \( MaxTS_{0:ml} \) so whenever it finds a timestamp greater than its own it changes the current epoch label because is know that something going wrong. Note that the canceling timestamp usually is empty and an empty timestamp is always greater than any given timestamp \((\perp <_e \forall i, t_i)\).

The writer maintains also a FIFO epoch queue wherein it saves the most recent timestamps. Any time it enqueues a timestamp that is already in the queue this timestamp is pushed to the front of the queue. The epoch queue has a maximum size so if this size is reached then the oldest timestamp in the queue must be dequeued. Note that the size of the queue is proportional to the number of epoch labels that can be stored in a system configuration. In other words is proportional to the total capacity of the communication links, namely \( O(cn^2) \) where \( c \) is the bound of the capacity of each link expressed in number of messages and \( n \) is the number of processors.

Note that the number of bits required to represent an epoch label depends on \( m \), the maximal size of the epoch queue, and it is in \( O(m \log m) \). Given that \( m \) is in \( O(cn^2) \) each epoch label requires \( O((cn^2 \log n + \log c)) \) bits. Therefore the size of a timestamp is the sum of the size of the epoch label and \( \log r \).

### 3.3.5 Algorithm’s Operations

In this subsection we present the main operations of the SelfS-ABD. Firstly, we list the two operations used for accessing the register, the QuorumWrite and QuorumRead operations.
After that, we described the read and write operations. Note that each read and write operation requires $O(n)$ messages and the size of the messages is linear in the size of a timestamp. Recall that in the present case, a quorum represents a majority of processors.

### 3.3.5.1 QuorumWrite Operation

This operation can be executed as part of both read and write operations. In a write operation the writer uses it to inform the system’s processor about the change it made so to maintain consistency between replicas. Similarly, in a read operation the reader uses it to inform the system’s processor about the maximum value it found and so to ensure the atomicity of reads.

The QuorumWrite operation sends a value $(t_{\text{new}}, v_{\text{new}})$ to every processor to be written to its local register and waits from a majority of acknowledgements. After that it terminates (see Figure 16). Note that an acceptable acknowledgement must have the same timestamp value with the current QuorumWrite timestamp $t_{\text{new}}$.

![Figure 16: QuorumWrite Operation Basic Steps](image)

The writer $p_0$ may receive a QuorumWrite$(t_{\text{new}}, v_{\text{new}})$ request which was sent as a result of a read operation. Whenever this happens, the writer checks if the request contains a timestamp
that is not equal to its local $MaxTS_{p, ml}$. If this is true then the writer $p_0$ enqueues $t_{new}$ in the epoch queue. (see Figure 16). Eventually the writer learns all the timestamps in the system which are not the same with its own local $MaxTS_{p, ml}$ timestamp (the proof is given in [3]).

Upon a request of QuorumWrite $(l, v)$
1: if $l \neq MaxTS_{0, ml}$ then enqueue($epochs, l$)

Figure 17: Pseudocode for receiving a QuorumWrite message by writer [3]

A reader $p_i$ may receive a QuorumWrite$(t_{new}, v_{new})$ request which was sent as a result of a read or write operation. Whenever a reader $p_i$ receives such that request it checks if the timestamp $t$ contained in the request is greater than its own timestamps, namely it checks if both $MaxTS_{i, ml} \prec_e t_{new}$ and $MaxTS_{i, cl} \prec_e t_{new}$. If this is true then $p_i$ adopts the receiving values because they are associated with a more recent update of the register. Specifically, $p_i$ assigns $MaxTS_i := \langle t_{new}, \bot \rangle$ and $v_i := v_{new}$. If not, $p_i$ assigns $MaxTS_{i, cl} := t_{new}$ only if $t_{new} \not\succ_e MaxTS_{i, ml}$ and $MaxTS_{i, ml} \neq t_{new}$. In this manner, the reader indicates that it has found a possible corrupted timestamp in the system. The pseudocode for that appears in Figure 17.

Upon a request of QuorumWrite $(l, v)$
1: if $MaxTS_{i, ml} \prec_e l$ and $MaxTS_{i, cl} \prec_e l$ then
2: $MaxTS_i := \langle l, \bot \rangle$
3: $v_i := v$
4: else if $l \not\prec_e MaxTS_{i, ml}$ and $MaxTS_{i, ml} \neq l$
   then $MaxTS_{i, cl} := l$

Figure 18: Pseudocode for receiving a QuorumWrite message by a reader [3]

3.3.5.2 QuorumRead Operation

This operation can be executed as part of both read and write operations. In particular it’s the first step of these operations.
When the writer $p_0$ performs a write operation, it first sends a *QuorumRead* request to the system’s processors to find out their local register variables. The writer is the only node in the system that can increase the value of $MaxTS.ml$. So by using the *QuorumRead* operation it can find whether there exists in the system a timestamp greater than its own. If this is the case, it changes the current epoch label because it knows that something went wrong (maybe occurs a transient fault).

Similarly, the first step of a read operation is the execution of a *QuorumRead*. By this way a reader $p_i$ can discover the local register variables of the rest of the processors in the system so that to find whether there is a maximal value in the system.

The *QuorumRead* operation sends a request to every processor for reading its local register variable and waits responses from a quorum (majority). Finally it returns the receiving values. (see Figure 19)

![QuorumRead Operation Basic Steps](image)

*Figure 19: QuorumRead Operation Basic Steps*

Whenever a processor $p_i$ receives a *QuorumRead* request, it returns back to the sender its local register variable value, $<MaxTS, v_i>$. 
3.3.5.3 Writer’s Operation

In the same way with the original algorithm ABD, process $p_0$ is the only writer in the system, and it is responsible for renewing the register’s value. Moreover, in the present approach the writer is liable to restore the system to a stable state after the occurrence of a transient fault.

Consequently the writer $p_0$ can change the current register’s value by performing a write operation. The pseudocode of this operation appears in Figure 20. Also in Figure 21 and Figure 22 appear the basic steps of the write operation.

In a write operation the writer firstly sends a QuorumRead request to the system processors to find out their local register variable values. In this way the $p_0$ can discover the timestamps that know the other processors of system and so to create a new timestamp greater than all.

After receiving answers $<\text{MaxTS}_1, \nu_1>$, $<\text{MaxTS}_2, \nu_2>$, ... from a quorum, the writer generates a new timestamp, $t_{\text{new}}$, which is greater than its local timestamp $\text{MaxTS}_{0,ml}$ and all received $\text{MaxTS}_j$ by one. After that the $p_0$ assigns the new value and timestamp as its local register variable values; $\text{MaxTS}_{0,ml} := t_{\text{new}}$ and $\nu_0 := \nu_{\text{new}}$. Finally it calls the QuorumWrite function with parameters $<\text{MaxTS}_{0,ml}, \nu_0>$, to inform the rest of the processors about the change it has made.

![Figure 20: Pseudocode of SelfS-ABD Write operation [3]](image-url)
**Round 1:**
Discover existing timestamps in whole system and then generate a new timestamp greater than all of them.

1. QuantumRead()
2. Message receivers send back their local register variable values
3. Wait $\text{ACK}_\text{QR}(\text{Max}TS, v_i)$ from majority
4. Generate a new timestamp, $t_{new}$ which is greater than timestamp $\text{Max}TS_{ml}$ and all received $\text{Max}TS_j$ by one

---

**Round 2:**
Propagate the new register value

5. Assign $\langle t_{new}, v_{new} \rangle$ as local register value
6. QuantumWrite($\text{Max}TS_{ml}, v_j$)
7. Message receivers perform the appropriate actions
8. Wait $\text{ACK}_\text{QW}$ from majority
9. Write Operation Completed

---

*Figure 21: SelfS-ABD Write Operation Basic Step – Round 1*

*Figure 22: SelfS-ABD Write Operation Basic Step – Round 2*
It is important to note that before it computes the new timestamp, $t_{\text{new}}$, the writer enqueues the received $ml$ timestamps and not empty $cl$ timestamps which are not equal to its local timestamp $MaxTS_0.ml$ to its epoch queue.

If its local timestamp label $MaxTS_0.ml.\ell$ is the greatest in the epoch queue (with respect to $\prec_b$), then it calls $\text{Next}_c$ to calculate a new timestamp. If the sequence number $MaxTS_0.ml.i$ is less than $r$, then the $\text{Next}_c$ increments it by one, leaving the $MaxTS_0.ml$ unchanged; so the timestamp that returns is $\prec MaxTS_0.ml.\ell$, $MaxTS_0.ml.i++$. Otherwise, if the writer’s local timestamp label isn’t the greatest in the epoch queue, it enqueues its local timestamp $MaxTS_0.ml$ to the epoch queue and applies $\text{Next}_b$ to obtain an epoch label greater than all the ones in its epoch queue. In this case the new timestamp is made by this epoch label and a zero sequence number, $t_{\text{new}} := <\text{Next}_b(\text{epochs}), \perp>$.

3.3.5.4 Reader’s Operation

A reader processor $p_i$ can perform read operations in order to find out the most recent value in the system, namely the value with the greatest timestamp. The pseudocode of this operation appears in Figure 23.

![Figure 23: Pseudocode of SelfS-ABD Read operation [3]](image)

A read operation of a reader $p_i$ begins with a $\text{QuorumRead}$. This allows the reader to discover the local register variables of the rest processors in the system. After receiving answers
\(<\text{MaxTS}_1, \upsilon_i>, \langle\text{MaxTS}_2, \upsilon_2\rangle, \ldots\) from a quorum, in the present case from the majority, the reader \(p_i\) tries to find the greatest timestamp, \(m_{l\text{max}}\), among its local \(\text{MaxTS}_i\) and all received \(\text{MaxTS}_j\) according to the comparison predicate \(<\nu\rangle.\) If such \(m_{l\text{max}}\) exists the \(p_i\) assigns \(\text{MaxTS}_i := \langle m_{l\text{max}}, \bot \rangle\) and \(\upsilon_i := \upsilon_{\text{max}}.\) To ensure atomicity of reads the reader \(p_i\) calls the \textit{QuorumWrite} function with parameters \(\langle m_{l\text{max}}, \upsilon_{\text{max}}\rangle\) before updates its local register variable values. In Figure 24 and Figure 25 appear graphically the steps of the \textit{read} operation.

**Figure 24:** SelfS-ABD Read Operation Basic Steps – Round 1

**Figure 25:** SelfS-ABD Read Operation Basic Steps – Round 2
3.3.6 Repeatedly Information Diffusion

Note that the authors of [3] consider an underlying stabilizing data-link protocol which is used for repeatedly diffusing the value of MaxTS from one processor to another. This helps the writer to learn more quickly a superset of the existing timestamps and so to generate a new one which will be the greatest in whole system faster. In the same time it helps a reader to discover whether there exists an unknown to it timestamp in the system which is larger than its own local timestamp.

In the sequel we use the term gossip message for referring to the message that is used for diffusing the value of MaxTS from one processor to another.

Whenever the writer $p_0$ receives a gossip message from a processor $p_j$ containing its local MaxTS, it checks if MaxTS$_0$.ml or MaxTS$_j$.cl is not equal to MaxTS$_0$.ml. If one of these is true then the writer $p_0$ enqueues the corresponding timestamp in the epoch queue. In other words, $p_0$ enqueues every diffused value that is different from MaxTS$_0$.ml. By this way the writer eventually learns all the timestamps which are not equal to its own MaxTS$_0$.ml. The pseudocode for this operation appears in Figure 26.

```
Upon a request of Gossip <MaxTS$_j$.ml, MaxTS$_j$.cl>
1: if MaxTS$_j$.ml $\neq$ MaxTS$_0$.ml then enqueue(epochs, MaxTS$_j$.ml)
2: if MaxTS$_j$.cl $\neq$ MaxTS$_0$.ml then enqueue(epochs, MaxTS$_j$.cl)
```

*Figure 26: Pseudocode for receiving a Gossip message by writer [3]*

Furthermore, if the MaxTS$_i$.cl of a reader $p_i$ is empty and $p_i$ receives from a processor $p_j$ a MaxTS$_j$ such that MaxTS$_j$.ml $\succ_e$ MaxTS$_0$.ml, then $p_i$ assigns MaxTS$_i$.cl := MaxTS$_j$.ml. Also
when $MaxTS_j, cl \prec_e MaxTS_i, ml$ then $p_i$ assigns $MaxTS_i, cl := MaxTS_j, cl$. The pseudocode for this operation appears in Figure 27.

\begin{figure}[h]
\begin{center}
\begin{minipage}{0.5\textwidth}
\begin{verbatim}
Upon a request of Gossip \langle MaxTS_j, ml, MaxTS_j, cl \rangle
1: if MaxTS_i, cl = ⊥ then
2: if MaxTS_j, ml \prec_e MaxTS_i, ml and MaxTS_i, ml \neq MaxTS_j, ml
3: then MaxTS_i, cl := MaxTS_j, ml
4: if MaxTS_j, cl \preceq_e MaxTS_i, ml and MaxTS_i, ml \neq MaxTS_j, cl
5: then MaxTS_i, cl := MaxTS_j, cl
\end{verbatim}
\end{minipage}
\end{center}
\caption{Pseudocode for receiving a Gossip message by reader [3]}
\end{figure}
Chapter 4

Implementation of the Algorithms

In this chapter we describe the key points of the implementation of algorithm ABD and SelfS-ABD. Both algorithms were developed in a Linux based environment and the implementations were written in the C programming language. Also, C sockets were used for interfacing with the TCP/IP communication network.

4.1 Establishing System Communication Network

Recall that both algorithms are emulating a SWMR atomic register in an asynchronous message-passing system which consists of $n$ processors. These processors are located at the nodes of a network and communicate by sending messages over communication links.

Put differently, a processor is a process that is running on a specific server and it is associated with a specific port number in which other processes can send connection requests. In order to do this each process must create a TCP/IP socket [15] and bind it to its host IP address and corresponding port number. After that it is ready to listen for new connections. In the present case, we used TCP/IP sockets, since the algorithms require reliable communication links.

In addition, the locations of all processors are known in advance. Every processor maintains a file called "host_list.txt" which contains the hostname of the servers as well as the
corresponding port number for all system processors. Note that every processor must have a unique port number so that to distinguish it from the rest.

Moreover, every processor holds a table with all the necessary information for each participant. This information is used for establishing the communication network of the system. As mentioned above, every pair of processors is directly connected via a unique reliable bidirectional communication link.

A processor \( p_i \) can send and accept connection requests. For every new connection, \( p_i \) creates a new TCP/IP sockets exclusively dedicated to this connection. Before it sends a connection request, \( p_i \) creates a new TCP/IP socket and the communication request is associated with that new socket. In the same way, when \( p_i \) accepts a new communication request, a new TCP/IP socket is created atomically. Each processor maintains a table `sockets[]` which contains all these sockets.

The position of a processor \( p_i \) in the table `processor_info_table[]` is used to determine how many requests should be sent and how many to accept while establishing the communication network. In particular, a processor \( p_i \) must send connection requests to all processors that are before it in the `processor_info_table[]` and must accept connection from the rest of the processors in that table. For example, the processor \( p_4 \) must send connection requests to processors \( p_0 - p_3 \) and accept connections from the rest, namely from \( p_5 - p_n \).

![Figure 28: Establish Communication Network](image)
An abstract algorithm pseudocode for this operation appears in Figure 29.

```c
// Function setupConnectionNetwork() - setups system TCP/IP network
void setupConnectionNetwork()
{
    ...
    // get current processor position
    current_pos = find_processor_position();
    // send connection request
    for (every processor j in positions 0 up to current_pos-1)
    {
        // create new socket fd
        sockets[j] = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        // bind the new socket to local processor address
        processorBind(sockets[j]);
        // connect with a processor j
        connect(sockets[j], processor_info_table[j].ptr_addr, processor_info_table[j].addr_length);
    }
    // accept connection request
    for (every processor j in positions current_pos+1 up to MAX_PROCESSORS)
    {
        sockets[j] = acceptConnection(listen_socket);
    }
    ...
}
```

**Figure 29: Pseudocode for Establishing Communication Network**

### 4.2 Executing Operations

As already mentioned, both algorithms implement a SWMR register where the processor $p_0$ is the writer and can perform write operations while the rest of the processors are readers and they can perform read operations. Moreover, at any time all processors must be able to receive and process incoming messages.

The default operation of all processors is to wait for new incoming messages while read or write operations can be performed after a preset interval. In particular, the writer $p_0$ may execute a write operation every $w\text{Int}$ seconds and a reader $p_i$ may execute a read operation every $r\text{Int}$ seconds.
Function unsigned alarm(unsigned seconds) [20] is used to determine when a read/write operation must be executed. This function causes the system to generate a SIGALRM signal after the number of second, specified by the seconds parameter, have elapsed. This signal is forcing the processor to interrupt the "listening for new messages" operation and perform a read or a write operation. When the new operation is completed the processor continues to wait for new incoming messages, namely it continues to executing the "listening for new messages" operation until the next alarm interrupt. For instance, the writer $p_0$ is used the alarm(wInt) to specify when to execute a write operation (Figure 30). Note that if seconds is zero every previously set alarm request is cancelled.

```
#include <unistd.h>
#include <signal.h>
...
// time interval between two write operations
#define wInt 6.4
...
/* Function signalAlarm() - automatic perform operation */
void signalAlarm(int sig){
    performWriteOperation();
    alarm(wInt);
}
...
/* Function main() */
int main(int argc, char *argv[]){
    ...
    // install alarm signal handler - automation read/write operations
    signal(SIGALRM, signalAlarm);
    ...
    // set alarm
    alarm(wInt);
    // program main job => wait messages
    process_listening();
    ...
}
```

*Figure 30: alarm(wInt) Example*

A processor $p_i$ is directly connected with each other processor $p_j$ in the system so it must be able to manage all these connections at the same time. In other words, because every connection is associated with an individual socket the given processor must handle multiple sockets at the same time. This can be done using the function select() [21]. This function allow a program to monitor multiple sockets, waiting until one or more of them become
"ready" for reading data, writing data as well as which of them have thrown an exception. Note that a socket is considered "ready" if it is possible to perform one of these operations without blocking.

In the present case select() is used to determine whether there are any data for reading in one or more sockets. If there exists such a socket, the processor receives the incoming message for that socket and then processes it. An abstract pseudocode appears in Figure 31.

```c
// Function process_listening() - listens for incoming messages
void process_listening(){
    ...
    fd_set readset;
    int maxfd = -1;
    ...
    while(1){
        // Initialize readset
        FD_ZERO(&readset);
        ...
        // add all of the interesting fds to readset
        for ( all sockets i in sockets[] ) {
            if(sockets[i] != CLOSE_SOCKET){
                if (sockets[i] > maxfd)
                    maxfd = sockets[i];
                FD_SET(sockets[i], &readset);
            }
        }
        ...
        // wait until one or more fds are ready to read
        select(maxfd+1, &readset, NULL, NULL, NULL);
        ...
        // process all of the fds that are still set in readset
        for ( all sockets i in sockets[] ) {
            if (FD_ISSET(sockets[i], &readset)) {
                ...
                // receive a TCP message
                recv(sockets[i], recv_msg, MAX_MSG_SIZE, 0);
                // process incoming message
                processIncomingMessage();
                ...
            }
        }
    }
}
```

Figure 31: select() Pseudo Code

The read and write operations of algorithm ABD are described in Sections 3.1.2 and 3.1.3. Also the corresponding operations of SelfS-ABD are described in Sections 3.3.5.3 and 3.3.5.4.
4.3 Terminating Execution

Under normal circumstances the algorithms are executed forever. However, the user can force them to terminate by pressing Ctrl-Z, Ctrl-C or Ctrl-\.

In addition both algorithms allow a minority of processors to crash, namely at any time must be accessible a majority of processors. If this is not the case then the respective processor terminates its execution automatically.

4.4 ABD Communication Protocol

The communication protocol of algorithm ABD supports the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Write Request</td>
<td>The writer informs a reader about the new value of the register.</td>
</tr>
<tr>
<td>2 Acknowledge to a Write Request</td>
<td>A reader informs the writer about the successful receiving of a write request message.</td>
</tr>
<tr>
<td>3 Read Request</td>
<td>A reader queries a processor about the current value of its local register replica.</td>
</tr>
<tr>
<td>4 Answer to a Read Request</td>
<td>A reader informs the sender about the current value of its local register replica.</td>
</tr>
<tr>
<td>5 Information Request</td>
<td>A reader informs a processor about the</td>
</tr>
</tbody>
</table>
maximum value of the register that just found.

A processor informs the sender about the successful receiving of an information request message.

Table 1: ABD Communication Protocol Operations

An extended description of these operations can be found in the Section 3.1

4.5 SelfS-ABD Communication Protocol

The communication protocol of SelfS-ABD algorithm supports the operations that are listed at Table 2. An extended description of these operations can be found in Section 3.3.5

Table 2: SelfS-ABD Communication Protocol Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 QuorumWrite Request</td>
<td>A processor informs a processor about the new value of its local register replica.</td>
</tr>
<tr>
<td>2 Acknowledge to a QuorumWrite Request</td>
<td>A processor informs the sender about the successful receiving of a QuorumWrite request message.</td>
</tr>
<tr>
<td>3 QuorumRead Request</td>
<td>A reader queries a processor about the current value of its local register replica.</td>
</tr>
<tr>
<td>4 Answer to a Read Request</td>
<td>A reader informs the sender about the current value of its local register replica.</td>
</tr>
</tbody>
</table>
In this chapter we present the experimental evaluation of the algorithms. We first provide an overview of the PlanetLab platform, and then we describe the parameters and the scenarios used for the evaluation. Then we present the results obtained from the experiments and we analyze the major findings.

5.1. The PlanetLab Platform

PlanetLab [7] is an open planetary-scale research network platform that supports the development of new network services. One of its main purposes is to serve as a testbed for overlay networks.

It was established by Larry Peterson (Princeton) and David Culler (UC Berkeley and Intel Research) on March 2002 [22]. The first version of it came on-line at the beginning of 2003 and since then more than 1,000 researchers at top academic institutions and industrial research labs have used it to develop new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing.

These days PlanetLab is composed from the PlanetLab Central and PlanetLab Europe [23]. As of May 2014, PlanetLab Europe consists of 348 nodes at 172 sites and at the same time
PlanetLab Central consists of 700 nodes at 418 sites [24]. Hence PlanetLab currently consists of 1048 nodes at 590 sites. Note that our University has access to PlanetLab Europe (OneLab).

![PlanetLab Europe map](image)

*Figure 32: PlanetLab Europe [23]*

PlanetLab provides a real asynchronous environment which is extremely unpredictable. Nodes may crash or reboot at any time and the communication delay varies. Moreover unexpected failures may happen anytime. A node may fail during a program execution and thereby it's possible to return incorrect results. Thus there is no guarantee regarding the nodes’ reliability. Also the communication between entities of a system can be interrupted. Finally the conditions of the network may change dramatically from one moment to another. Hence, this is an ideal environment to test the reliability of Fault-Tolerant and Self-Stabilizing algorithms!
5.2. Experimental Platform

The experimental environment consisted of a set of processors, one writer and many readers. For the experiments we used following 20 physical machines (Table 3).

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>planetlab1.unineuchatel.ch</td>
<td>Switzerland</td>
</tr>
<tr>
<td>planetlab2.diku.dk</td>
<td>Denmark</td>
</tr>
<tr>
<td>planet1.13s.uni-hannover.de</td>
<td>Germany</td>
</tr>
<tr>
<td>plab4.ple.silweb.pl</td>
<td>Poland</td>
</tr>
<tr>
<td>planetlab4.hiit.fi</td>
<td>Finland</td>
</tr>
<tr>
<td>planetlab2.bgu.ac.il</td>
<td>Israel</td>
</tr>
<tr>
<td>planetlab4.inf.ethz.ch</td>
<td>Switzerland</td>
</tr>
<tr>
<td>planetlab-3.imperial.ac.uk</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>peeramidion.irisa.fr</td>
<td>France</td>
</tr>
<tr>
<td>planetlab2.cesnet.cz</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>planetlab1.urv.cat</td>
<td>Spain</td>
</tr>
<tr>
<td>pli1-pa-4.hpl.hp.com</td>
<td>United States</td>
</tr>
<tr>
<td>host1.planetlab.informatik.tudarmstadt.de</td>
<td>Germany</td>
</tr>
<tr>
<td>planetlab2.unineuchatel.ch</td>
<td>Switzerland</td>
</tr>
<tr>
<td>planetlab2.urv.cat</td>
<td>Spain</td>
</tr>
<tr>
<td>planet2.13s.uni-hannover.de</td>
<td>Germany</td>
</tr>
<tr>
<td>planetlab3.inf.ethz.ch</td>
<td>Switzerland</td>
</tr>
<tr>
<td>planetlab-node1.it-sudparis.eu</td>
<td>France</td>
</tr>
<tr>
<td>planet-plc-5.mpi-sws.org</td>
<td>Germany</td>
</tr>
<tr>
<td>mercury.silicon-valley.ru</td>
<td>Russia</td>
</tr>
</tbody>
</table>

Table 3: List of machines that hosted the experiments

Each of these machines was used to host one or more processors. If the number of processors exceeded the number of hosts then each host was running more than one processes instances in a round robin way. For example, if the number of system processors was 25, then on the first 5 machines we were running two process instances while in the remaining 15 machines only one.
In addition, we used the positive time parameters $w_{\text{Int}} = 8\text{sec}$ and $r_{\text{Int}} = 10\text{sec}$ to model operation frequency. Specifically the writer executed a write operation every $w_{\text{Int}}$ seconds while the readers randomly picked a time between $[6...r_{\text{Int}}]$ to execute their next read operation. Note that each reader chose a new time interval every time its latest operation was completed.

To determine the chosen parameter, we measured the average time that needed for each participant to complete its operation. Then we chose the greatest among them which was approximately 0.4 seconds. Due the asynchronicity of the experimental environment this value was increased up to those values in order to allow sufficient time to all processes to complete their operation. Additionally, as already mentioned, in each operation (either write or read) a processor should expect responses from a majority. Thus, by allowing the readers to randomly select the time interval between their operations, it increases the probability of receiving responses from different processors.

Furthermore each participant performed exactly 200 operations. Namely in each experiment the writer performed up to 200 write operations and the readers performed exactly 200 read operations. We chose this upper bound according to previous works related to this topic [25] [26].

Finally to simplify the complexity of algorithm SelfS-ABD the size $k$ of the Antistings was set equal to the number of system processors, namely $k = n$ and thus the size $\mathcal{K}$ of the set $\mathcal{X}$ was defined as $n^2 + 1$ ($\mathcal{K} = n^2 + 1$). Also the length of the epoch queue was set as $n$ and the sequence number upper bound $r$ of the timestamp was defined as 2000.

As already mentioned, the writer may increase the sequence number of its current timestamp and if this sequence number already equals to the upper bound $r$ then it changes the epoch
label of its timestamp. By choosing \( r = 2000 \) the writer is "forced" to change the epoch label of its timestamp when and only when it discovers an epoch label larger than its own. Under normal circumstances the sequence number will never reach this upper bound. In the current experiments ideally the writer will generate from the first write operation an epoch label that will remain the same until the end of the simulation. Thus because the writer can execute 200 write operations the maximum value of the sequence number will be, in the best case, 200.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors Count</td>
<td>( n )</td>
</tr>
<tr>
<td>Antistings size ( k )</td>
<td>( n )</td>
</tr>
<tr>
<td>Set ( \mathcal{X} ) size ( \mathcal{K} )</td>
<td>( n^2 + 1 )</td>
</tr>
<tr>
<td>Epoch Queue Length</td>
<td>( n )</td>
</tr>
<tr>
<td>Sequence Number Upper Bound ( r )</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 4: SelfS-ABD Experimental Parameters

5.3. Experimental Scenarios

The experimental evaluation focused on the following three scenarios:

**Scenario 1: Overhead due to Self-Stabilization:** This scenario was designed to test the overhead due to self-stabilization as the number of processors’ increases. In particular, we executed both algorithms with 10, 20 and 30 processors so as to test their performance and scalability.

**Scenario 2 - SelfS-ABD Behavior:** This scenario was designed to test the behavior of algorithm Self-ABD with one corruption at the beginning of the simulation. As mentioned above, the algorithm Self-ABD allows the system processors to start at an arbitrary state so
this scenario was designed to test whether the system can eventually stabilize and reach a legitimate state.

**Scenario 3 – Information Diffusion Effect:** This scenario was designed to test whether the use of the information diffusion (gossip protocol) affects the behavior of algorithm SelfS-ABD. Specifically has been tested if the use of the gossip protocol changes the results of the previous two scenarios.

5.4. Experimental Results

**Scenario 1: Overhead due to Self-Stabilization**

Aforementioned in Section 5.3, this scenario was designed to test the overhead due the self-stabilization as the number of processors’ increases. For this reason we performed 3 experiments with 10, 20 and 30 processors respectively. Each experiment was executed five times so that to remove those with the largest and smallest operation latency. The final results are the average latency of the remaining three executions. Note that the operation latency was measured as the time needed for an operation to complete, namely the time from its invocation until its completion. Here we did not make use of information diffusion (gossip).

Table 5 presents the average operation latency of the writer. Also Graph 1 shows the average operation latency of the writer in milliseconds as the number of processors increases. It can clearly be seen that the average operation latency of the writer in algorithm SelfS-ABD is considerably larger than that of ABD.
Number of Processors | ABD Average Writer Operation Latency (msec) | SelfS-ABD Average Writer Operation Latency (msec)
--- | --- | ---
10 | 26,047 | 55,256
20 | 27,274 | 57,375
30 | 28,533 | 58,708

Table 5: Scenario 1 Results - Average writer operation latency in msec

Graph 1: Scenario 1 - Writer Operation Latency

According to algorithm ABD, whenever the writer wants to perform a write operation it increases its current sequence number, sends a write message to rest processors to inform them about the new value of the register and after that waits for majority acknowledgments. In contrast, at algorithm SelfS-ABD the writer first sends a QuorumRead request to the rest of the system’s nodes to find out their local register variables and after receiving answers from a majority it generates a new timestamp which is greater than its local timestamp and all received timestamps by one.

So in the SelfS-ABD algorithm the write operation requires the writer to contact with the rest twice: first to find out their local timestamps and after to inform them about the new value of
the register while in the ABD the writer communicates only to inform the others about the new value of the register.

Moreover as described in detail in Subsection 3.3.5.3 the generation of a new timestamp in algorithm SelfS-ABD is more complex and involves more computational steps than in algorithm ABD, where the writer needs to increase only the sequence number.

Therefore the *write* operation in SelfS-ABD is more time consuming than in the algorithm ABD because SelfS-ABD requires more computations so that the system to be self-stabilizing.

Table 6 presents the average operation latency of the readers.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>ABD Average Readers Operation Latency (msec)</th>
<th>SelfS-ABD Average Readers Operation Latency (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>75,661</td>
<td>49,451</td>
</tr>
<tr>
<td>20</td>
<td>86,644</td>
<td>55,094</td>
</tr>
<tr>
<td>30</td>
<td>91,049</td>
<td>58,778</td>
</tr>
</tbody>
</table>

*Table 6: Scenario 1 Results - Average readers operation latency in msec*

Graph 2 depicts the average operation latency of the readers in milliseconds as the number of processors increases. It is noteworthy that the average operation latency of the readers in algorithm SelfS-ABD is smaller than in that ABD.
According algorithm ABD, a reader can perform read operations in order to find out the most recent value in the whole system, namely the value with the highest sequence number. To do so it first queries all processors to find out their local register value and after receiving answers form a majority it picks the value with the highest sequence number. Then it propagates the value that it picked. Likely, in algorithm SelfS-ABD a reader first sends a QuorumRead request to the rest to find out their local register variables and after receiving answers from majority it tries to find the greatest timestamp among its local timestamp and all received timestamps according to the comparison predicate $\prec$. If such a timestamp exists it calls the QuorumWrite function so as to propagate the value it found.

In both cases a reader must communicate with the rest of the processors twice: first to find out their local register replica and after to propagate the maximum value that found. In the algorithm ABD a reader can always find the value with the highest sequence number and hence always communicates with the others twice but in the case of algorithm SelfS-ABD the reader may abort if it there exist no such a maximal value and consequently it communicates with the others only once. This is a possible explanation of why the algorithm SelfS-ABD appears to have better read operation latency.
It is noteworthy that average operation latency of the readers in algorithm ABD is twice larger than the average operation latency of the writer in that algorithm. This was expected as in the algorithm ABD a reader communicates with the others twice but in the case of write operation the writer communicates with the others only once.

Furthermore, the average operation latency of the writer and readers, in both algorithms, grows as the processors’ count increases. The more processors participate, the more messages are sent in the network. Hence there is more congestion in the communication network and this leads to greater delay of messages delivery. In the same time, the writer has to wait acknowledgments from more processors as the majority changes due the change of the number of processors.

From of the above, it appears that self-stabilization adds additional time to the algorithm. In the self-stabilizing version, the writer is the one that is responsible for the register value and also for restoring the system to a legitimate state after the occurrence of a transient fault. To be able to do so is required to perform extra computations which lead to the increase of the write operation’s latency.

At the same time, the space complexity of algorithm SelfS-ABD is dramatically greater than that of ABD. Indicatively, in algorithm ABD the register value is a pair of \(<seq\_num_i, \upsilon_i>\) where \(seq\_num_i\) is a positive integer number and \(\upsilon_i\) is the value of the register. However in SelfS-ABD, each processor \(p_i\) holds a variable \((MaxTS_i, \upsilon_i)\) where \(MaxTS_i\) is a pair \(<ml_i, cl_i>\) and \(\upsilon_i\) is the value of the register. Note that \(ml_i\) and \(cl_i\) of \(MaxTS_i\) are both timestamps. Therefore, in algorithm SelfS-ABD, as the number of processors increases, more bits are required in order to represent the timestamps that are associated with the register value since each timestamp depends on the size of the epoch label. Recall that an epoch label is a pair \((s, \mathcal{A})\) where \(s\) is an integer from a fixed set \(\mathcal{X}\) and \(\mathcal{A}\) is a subset with \(k\) elements of that set. In the present experiments \(k\) is equals to the number of processors hence as more the
processors are as larger is the $k$ and therefore are required more bits to represent an epoch label. Moreover, in SelfS-ABD, the writer must hold an epoch queue which increases even more the space complexity of the algorithm.

Additionally the size of the messages in SelfS-ABD is also greater than in ABD. Table 7 presents the maximum size of the messages that was used in each experiment.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>ABD Maximum Message Size (bits)</th>
<th>SelfS-ABD Maximum Message Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td>20</td>
<td>170</td>
<td>220</td>
</tr>
<tr>
<td>30</td>
<td>170</td>
<td>320</td>
</tr>
</tbody>
</table>

*Table 7: Scenario 1 Results – Maximum size of the messages in bits*

Graph 3 shows the maximum size of the messages in bits as the number of processors increases. It is immediately apparent that the maximum message size in algorithm ABD does not change while in algorithm SelfS-ABD, as expected, it grows as the number of processor increases.

*Graph 3: Scenario 1 – Maximum size of Messages*
In algorithm SelfS-ABD the largest message that can be sent by a processor is a read acknowledgment which contains its local register variable value. As already mentioned, as the number of processors increases, the number of bits required to represent the local register variable value increases too. For this reason the maximum message size in algorithm SelfS-ABD grows as the number of processors increases. But contrary, in algorithm ABD the register value is a pair $<seq_{num}, v_i>$ which is independent of the processors’ count. Hence the bits that are required to represent it are always the same.

**Scenario 2 - SelfS-ABD Behavior**

As mentioned in Section 5.3 this scenario was designed to test the behavior of algorithm SelfS-ABD with one corruption at the beginning of the simulation. Algorithm SelfS-ABD allows the system processors to start at an arbitrary state so this scenario was designed to test whether the system can eventually stabilize and reach a legitimate state.

Note that the system reaches a legitimate state when the local replicas of the registers of all processors are the same with the local register replica of the writer. In other words all readers must be holding the newest value of the register that was introduced by the writer. Here we did not make use of information diffusion (gossip).

For this scenario we used the same experiments as the previous one. Specifically we tested how many readers were had the same register timestamp with the writer at the end of the simulation in each of these experiments. Table 8 lists these results.
<table>
<thead>
<tr>
<th>Number of Readers</th>
<th>Experiment Number</th>
<th>Same Register Value with the Writer</th>
<th>Different Register Value from the Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>18</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24</td>
<td>5</td>
</tr>
</tbody>
</table>

*Table 8: Scenario 2 Results*

**Self-ABD Behavior with 9 Readers**

*Graph 4: Scenario 2 - Self-ABD Behavior with 9 Readers*
According to Graph 4, Graph 5 and Graph 6, at the end of each simulation at least the majority of the readers has the same register value with the writer.
Following the description of SelfS-ABD, at the beginning of a simulation each processor of the system has a different timestamp. Hence the writer must be aware of a superset of the existing timestamps so that to generate a new timestamp greater than all of them. As mentioned in Subsection 3.3.5.3 in a write operation the writer firstly sends a QuorumRead request to the rest processors to find out their local register variable values. In this way it can discover the timestamps of the other processors and so to create a new timestamp greater than all. However it waits answers from a majority, so it is possible not to know the timestamp of all processors in the system because some of them do not caught up to be in the majority of the answers.

Unfortunately, in the present experimental evaluation, at only one experiment (3rd experiment with 9 readers) the writer succeeded to know all the existing timestamps in the system and so to generate a new one greater than all and hence the system to reach a legitimate state.

According to [3], the convergence of the system is postponed until the writer is aware of a superset of the existing timestamps. At the most experiments this never happens because the simulation stops before the writer is made aware of a superset of the existing timestamps. For this reason the system does not reach its legitimate state.

There are many reasons that due to these results as such the communication delays, the network congestion as well as the workload of each processor at the time when the writer inquires to find out the local register replica of the others. Also the writer can perform only 200 writes operations which may not be enough.

As discussed in Section 3.3.6 the authors of [3] consider an information diffusion protocol which is used for repeatedly diffusing the value of register replica from one processor to another. However this protocol has not been used in the present scenario and that is the main reason why the system fails to reach a legitimate state until the end of the simulations.
To conclude in this scenario the atomicity ensured always but the system could not be stabilize.

**Scenario 3 – Information Diffusion Effect**

As mentioned in Section 5.3 this scenario was designed to test whether the use of the information diffusion (gossip protocol) affects the behavior of algorithm SelfS-ABD. Specifically has been tested if the use of the stabilizing data-link protocol changes the results of the previous two scenarios. Also, in this scenario we followed the same methodology with the above scenarios, namely we performed 3 experiments with 10, 20 and 30 processors respectively. Note that a gossip message was sent every 10 operations namely every 10 normal operations of a processor $p_i$ the 1 was the gossip. We chose this value so as the *gossip* operation not to dramatically affect the performance of algorithm SelfS-ABD.

We first tested how many readers were had the same register timestamp with the writer at the end of the simulation. Table 9 lists these results. It can clearly be seen that at the end of each simulation *all* the readers had the same register value with the writer.

<table>
<thead>
<tr>
<th>Number of Readers</th>
<th>Experiment Number</th>
<th>Convergence Time</th>
<th>Different Register Value from the Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>29</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 9: Scenario 3 Results*
Following the results of the Scenario 2 (Table 8) without the use of the gossip protocol almost in all executions, at least the majority of the readers has the same register value with the writer. Unlike with the gossip protocol the system can always be stabilized. The Graph 7 clearly shows that differentiation.

![Self-ABD Behavior](image)

*Graph 7: Scenario 2 Vs Scenario 3*

Consequently the gossip protocol significantly helps the writer to discover all the existing timestamps in the system and so to generate a new one greater than all and therefore the system to be stabilized, namely to reach always a legitimate state. This confirms the related discussion in [3], where it was argued that the gossip protocol should play an important role to convergence.

Table 10 presents the convergence time of the algorithm in each experiment. Specifically lists the number of the write operation during which the writer was generate a timestamp greater than all in the system. Also, Graph 8 shows graphically these results.
<table>
<thead>
<tr>
<th>Number of Readers</th>
<th>Experiment Number</th>
<th>Convergence time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>145</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

*Table 10: Scenario 3 Results – Convergence Time*

Graph 8 shows that when there are few processors in the system the convergence time of it, is smaller than that of other cases. However the number of the processors is not the only reason that affects the convergence time of the system because, according to our experiments, the convergence time of the system with 20 processors is must larger than that of 30 processors.
We now examine whether the use of the gossip protocol affects the average operation latency of the writer and the readers.

Table 11 presents the average operation latency of the writer including the results of the current experiment. Also, Graph 9 shows graphically these results.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>ABD Average Writer Operation Latency (msec)</th>
<th>SelfS-ABD without Gossip Protocol Average Writer Operation Latency (msec)</th>
<th>SelfS-ABD with Gossip Protocol Average Writer Operation Latency (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>26,047</td>
<td>55,256</td>
<td>55,753</td>
</tr>
<tr>
<td>20</td>
<td>27,274</td>
<td>57,375</td>
<td>56,753</td>
</tr>
<tr>
<td>30</td>
<td>28,533</td>
<td>58,708</td>
<td>58,488</td>
</tr>
</tbody>
</table>

*Table 11: Scenario 3 Results - Average writer operation latency in msec*

*Graph 9: Scenario 3 - Writer Operation Latency*
Graph 9 shows that the use of the gossip protocol does not affect the average operation latency of the writer. This was expected as the write operation is independent of gossip.

Table 12 gives the average operation latency of the readers. Also Graph 10 depicts the average operation latency of the readers in milliseconds as the number of processors increases. It can clearly be seen that the average operation latency of the readers in algorithm SelfS-ABD increases with the use of the gossip protocol.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>ABD Average Readers Operation Latency (msec)</th>
<th>SelfS-ABD without Gossip Protocol Average Readers Operation Latency (msec)</th>
<th>SelfS-ABD with Gossip Protocol Average Writer Operation Latency (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>75,661</td>
<td>49,451</td>
<td>73,067</td>
</tr>
<tr>
<td>20</td>
<td>86,644</td>
<td>55,094</td>
<td>75,537</td>
</tr>
<tr>
<td>30</td>
<td>91,049</td>
<td>58,778</td>
<td>78,7586</td>
</tr>
</tbody>
</table>

*Table 12: Scenario 3 Results - Average readers operation latency in msec*

*Graph 10: Scenario 1 - Readers Operation Latency*
According algorithm SelfS-ABD’s description, after the system stabilizes every *read* operation will always return the maximal value in the system. Hence, when is used the gossip protocol a reader must communicate with the rest of the processors twice: first to find out their local register replica and after to propagate the maximum value that found. In contrast, without the gossip protocol the system may not be stabilized so a *read* operation may be repeatedly aborted until the write writes new timestamps. For this reason the average operation latency of the readers with the stabilizing data-link protocol is greater than without it.

To conclude the major finding of this scenario is that the gossip protocol helps the system to stabilize. Therefore, in this scenario ensured always both atomicity and self-stabilization.
Chapter 6

Conclusion

This chapter summarizes the major conclusions obtained through the experimental evaluation of algorithms ABD and SelfS-ABD. We conclude with a discussion on future directions.

6.1 Conclusions

The major objective of this thesis was the implementation of algorithm SelfS-ABD [3]. This algorithm is a self-stabilizing version of the classical algorithm ABD [1] that permits automatic recovery of a system from any kind of transient errors. The system can start from an arbitrary state and it eventually exhibits its desired behavior after a finite number of steps without explicit human intervention.

To the best of our knowledge, algorithm SelfS-ABD is the first version of algorithm ABD that incorporates mechanisms for handling transient failures. In particular, to overcome the problems which may possibly appear due transient faults, the authors of [3] suggest splitting the emulation execution into epochs, periods during which the sequence numbers are supposed not to wrap around. Whenever a "corrupted" sequence number is found, a new epoch is begun, overriding all previous epochs. This procedure repeats until there aren't any more "corrupted" sequence numbers in the system, and the system stabilizes. After the system’s stabilization, the epoch remains the same at least until the sequence number wrap...
around again. Epochs are denoted with labels from a bounded domain using a new bounded label scheme introduced in [3]. This scheme safeguards that if the writer eventually learns about all the epoch labels in the system, then it will generate a new epoch label greater than all of them.

It is noteworthy that algorithm SelfS-ABD is quite complex and some parts of it are difficult to understand. Also the debugging of the algorithm implementation was quite challenging and time consuming, mainly due to the complex data structures required by the algorithm.

The second goal of this thesis was the evaluation the practicality of algorithm SelfS-ABD by deploying it on PlanetLab. We first compared its operation latency with that of classical algorithm ABD in order to investigate the additional overhead required for self-stabilization. According to our experimental evaluation, self-stabilization does add extra overhead to the algorithm. In algorithm SelfS-ABD the writer is responsible for restoring the system to a legitimate state after the occurrence of a transient fault and to be able to do that is required to perform extra computations which lead to the increment of the write operation latency. Moreover read operations may be repeatedly aborted if the set of timestamps does not include a timestamp greater than the rest. In this case the convergence of the system is postponed until the writer is aware of a superset of the existing timestamps.

In addition, self-stabilization increases the space complexity of the algorithm as described in detail in the first scenario of the experimental evaluation.

The major finding of the experimental evaluation is that the use of the gossip protocol significantly helps to the system stabilization. In particular it helps the writer to be quicker aware of a superset of the existing timestamps and so to generate faster a new one which will be the greatest in the system. Hence the system will stabilize much faster. Hence it appends that the use of this protocol is necessary for convergence.
When the writer is aware of all existing timestamps the results of reads and writes operations of algorithm SelfS-ABD can be totally ordered in a manner that respects the real-time order of non-overlapping operations, so that the sequence of operations satisfies the semantics of a SWMR register.

Finally, to the best of our knowledge our work is the first attempt to study the practicality of algorithm SelfS-ABD and to empirical evaluate its behavior.

### 6.2 Future Work

The experimental evaluation of algorithm SelfS-ABD focused on three scenarios. A more deeper and comprehensive experimental analysis could possibly help in the better understanding of the algorithm as well as its behavior in realistic environments.

It would be good in the future to implement the stabilizing data-link protocol the existence of which is supposed by the authors of [3]. Also can study the practically of the algorithm SelfS-ABD using this protocol.

Algorithm SelfS-ABD is developed for SMWR registers. So an open pending issue is the implementation of MWMR registers in message passing systems using the self-stabilization technique to overcome transient failures.
References


[22] PlanetLab History. [Online]. http://www.planet-lab.org/history
[23] PlanetLab Europe. [Online]. https://www.planet-lab.eu/


[26] Antigoni Xatzidimitriou, "Implementation and Experimental Evaluation of Algorithm SLIQ", Diploma Project, Department of Computer Science, University of Cyprus, 2009