IMPLEMENTATION OF A SELF STABILIZING RECONFIGURATION ALGORITHM WITH THE HELP OF COMMUNICATION LIBRARY ZeroMQ

Ioannis Aristidou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2018

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

IMPLEMENTATION AND EVALUATION OF SELF STABILIZATION ALGORITHM WITH THE HELP OF COMMUNICATION LIBRARY ZeroMQ

Ioannis Aristidou

Supervisor
Chryssis Georgiou

This thesis is submitted to the Undergraduate Faculty of University of Cyprus in partial fulfillment of the requirements for the Degree of Computer Science at Computer Science Department

May 2018

Acknowledgements

Special thanks to my supervisor Dr Chryssis Georgiou for giving me the opportunity to work on with this subject.

As my supervisor, he helped me organizing my schedule and giving me some papers to study so to gain the required for the project. I would like to thank him for the excellent corporation, his patience, his interest and for all of his help which was very crucial, especially at the moments when I had to face some difficulties. Furthermore special thanks to the Doctoral student Mr Ioannis Markoullis for his help which was very important for me.

Due to the fact that I had to face some difficulties which cost me time I did not manage to finish the whole project. Despite of tough lack I am so satisfied from my work because I have the chance to study a computer science’s domain which I did not know very well before. As a result this thesis helped me to extend my knowledge.

Abstract

The thesis presents the development of a self-stabilizing reconfiguration algorithm with the use of ZeroMQ communication library.

Self-stabilization is an important concept of fault tolerance in distributed computing. No matter which is the initial state of the system, it guarantees that in a finite number of steps the system will reach a legal state.

We consider distributed systems that work in dynamic asynchronous environments, such as a shared storage system. A configuration is a set of active computing processors (servers) participants which typically provide services to other participants of a system. As time passes, the system allows new arriving processors to join, but when an amount of participants leave the configuration or stop working due to failure, the system may need to reconfigure.

To achieve the above an algorithm consisting of three modules has been developed. The first one is called the Reconfiguration Stability Assurance which provides information on the current configuration and on whether a reconfiguration is taking place. The second module is called the Reconfiguration Management which evaluates whether a reconfiguration is needed The third module is a Joining Mechanism which allows, when there is no reconfiguration process taking place, new processors to join the system. The message exchanging between processors is done with the use of the ZeroMQ library.

Finally, a demo simulation has been implemented where two processors, as a part of the configuration, talk to each other and then another processor uses the joining mechanism to become a participant

Completing this thesis I learned a lot about distributed systems, self-stabilization and how to use the ZeroMQ library despite of the difficulties that I had to face. The knowledge that I gained from this work will be very useful in the near future.

Contents

Chapter 1

Introduction

1.1 Purpose and Motivation

In computer science, distributed systems are one of the most challenging areas of research. A distributed system [1] is a collection of autonomous independent computing elements, generally referred as node, that appears to user as a single coherent system. The above definition refers to two features of distributed systems. The first one is that each computing element which is a part of a distributed system is able to behave independently from the others. The second one is that due to the fact that users believe that they cope with a single system, with one way or another each element need to collaborate with the others. A node which belongs to a distributed system can be everywhere. Even geographically far away from the other nodes.

Distributed algorithms are executed from every node of a distributed system in parallel despite of where the node is located. All nodes are connected as a fully connected graph. As a result, they communicate with each other by exchanging messages but all decisions are taken locally. There are two types of these algorithms, synchronous and asynchronous. Synchronous are algorithms that are executed from nodes in a synchronized environment. For example, every node knows when to communicate with others and no one acts asynchronous. On the other hand asynchronous distributed algorithms are executed by nodes who act without synchronization. This makes tougher the work for a programmer in case when there is no good design and organization of the problem. As a result some execution problems may appear and will have negative effects on a distributed system. For instance in a short amount of time more processors may come to join the system and more processors may leave the system without synchronization.

In that case this type of system has the ability to recover. This ability is called self-stabilization. On distributed systems, self – stabilization [2] guarantees despite of current state of the system at any time due to an arbitrary fault, in a finite number of steps the system will converge to a legal state.

The purpose of this thesis is to implement a self-stabilizing reconfiguration algorithm and examine how much time it takes for a distributed system which works on asynchronous environment to recover after an arbitrary fault.

1.2 Methodology

At first I had a meeting with my supervisor who made me a brief introduction to the subject. After that he gave me some articles to study, which were an overview about the topic.

When I read the articles I started learning the ZeroMQ library by reading the official guide and its examples from the library’s official site. I installed the library on Debian 7 operating system, which I used through a virtual machine. The interface of Debian 7 is presented in Figure 1. I learned about this operating system from a friend who suggested to me to install it instead of Ubuntu due to the fact that it was easy the process of installation.

After that my supervisor asked me to create a small program on C using the ZeroMQ library. I had to make three processes to communicate with each other bidirectional.

On December I started the study of self-stabilizing reconfiguration paper to understand the three modules of the algorithm which I had to implement.

At the end I had to make some experiments with the algorithm to demostrate that it is working. The only scenario which I checked was when the system works fine and a new processor comes to join.

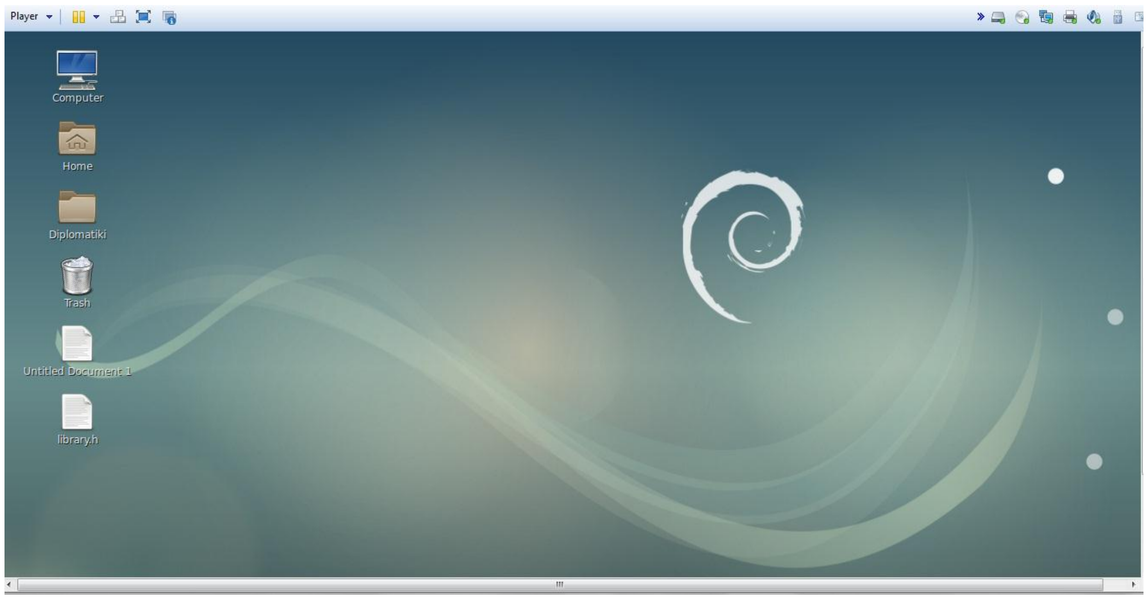


Figure 1. The interface of Debian 7 on VMware

1.3 Structure of Diploma Thesis

At Chapter 2, is presented the background of the thesis. At first there are some details about distributed systems. Specifically they are explained some characteristics of distributed systems as well as the middleware. Moving on, there is a description about the ZeroMQ library which is used for distributed messaging. Furthermore there is a description about self-stabilization ability and its relationship with distributed systems. Specifically in this subsection they are described two fundamental problems of distributed systems and how self-stabilization can solve them.

At Chapter 3, there is the description of self-stabilizing reconfiguration scheme. Specifically is presented the problem, some system settings and an explanation of how the three modules communicate each other.

At Chapter 4, is given more emphasis on the three modules. It is explained briefly what is the role of each module and then is described how the implementation was done.

Moving on, in Chapter 5 is presented an implementation example.

Finally, Chapter 6 is the conclusion part of the thesis. At first there are general conclusions for the project. Then they are described problems which I have to face during my work on the project and at section future work is given some details about how someone can use the three modules, which I implemented, in the near future. Also there are some suggestions which will enhance the performance of the algorithm. Finally there is a section about benefits which I gained from the thesis.

Chapter 2

Background

2.1 Distributed Systems

In this section there is a brief description about distributed systems and its characteristics. Also there is a reference on middleware which is a software which is typically used in distributed systems.

2.1.1 Definition

A distributed system is a collection of autonomous computing elements which are connected through a network and are distributed in any place geographically. It appears to users as a single coherent system.

2.1.2 Characteristic features of distributed systems

From this definition we note two significant characteristic features, as it described in [1] of a distributed system. The first one is that a distributed system is a collection of an amount of computing elements which are able to behave independently of each other. A computing element, which is referred as node, can be either a software process or a hardware device. The second one is that users believe that they are dealing with a single system. In other words autonomous nodes may have to collaborate with one way or another.

2.1.3 Characteristic 1: Collection of autonomous computing elements

According to [1], a fundamental principle is that nodes can act independently from each other. The fact is if they ignore other nodes in the system then, there is no used to be a part with others in it. As a result of this, all nodes are exchanging messages between them because they are programmed to achieve common goals.

To achieve communication between nodes, each node must know all members which it can communicate directly. Managing group membership is not an easy process. It depends on the type of the group, open or close, which we have in front of us. In an open group every node is allowed to join in the system. So it can send messages to everyone. On the other hand in a closed group, only the group members can communicate with each other and there is a separated joining mechanism which allows a node to join in the system. In that case a member from the system is protected from communicating with an intruder whose aim is to create havoc.

Practice shows that a distributed system is often organized as an overlay network. A node is a software process which is equipped with a list of processes where it can directly send messages to. Also there is a case where at first, the neighbor must be looked up. Message passing is done through TCP/IP or UDP channels (Peer-to-peer network). These cases are called Structured Overlay and Unstructured Overlay respectively.

In Structured Overlay all nodes are organized on a tree or a ring topology. This means that every node knows exactly its neighbors.

On the other side in Unstructured Overlay each node knows nothing about its neighbors. As a result, at first the neighbor must be looked up.

2.1.4 Characteristic 2: Single Coherent System

According to distributed system’s definition which is given in [1,2] , a distributed system appears to users as a single coherent system. This happens when the system behaves according to user expectations. All nodes operate the same. It does not matter the place, the time and when the interaction between user and the system is taking place. Single coherent view of the system is often challenging enough. For example, it requires that, an end user would not be able to tell exactly on which computer a process has to execute. Furthermore where data is stored should be of no concern and neither should it matter that the system may be replicating data to enhance its performance. This is called distribution transparency [1].

However, when we try to appear a distributed system as a single coherent system, we introduce an important trade-off. As we cannot ignore the fact that a distributed system consists of multiple, networked nodes, it is inevitable that at any time a part of the system fails due to unexpected behavior.

2.1.5 Middleware and Distributed Systems

A middleware is a separated layer which is placed on the top of each computer’s local OS and under the layer of the application as it is shown in Figure.2

Figure 2 shows where the middleware is located. This software is used in distributed systems to allow components of an application to communicate with them as well as an application can communicate with each other using the same interface.

Middleware [1] is the same as an operating system to a computer. It offers its applications to efficiently share and deploy those resources across a network. The main difference between middleware and operating system is that middleware services are offered in a networked environment.

Furthermore we can see middleware as a container of commonly used of components and functions which there is no need to be implemented from each application separately.

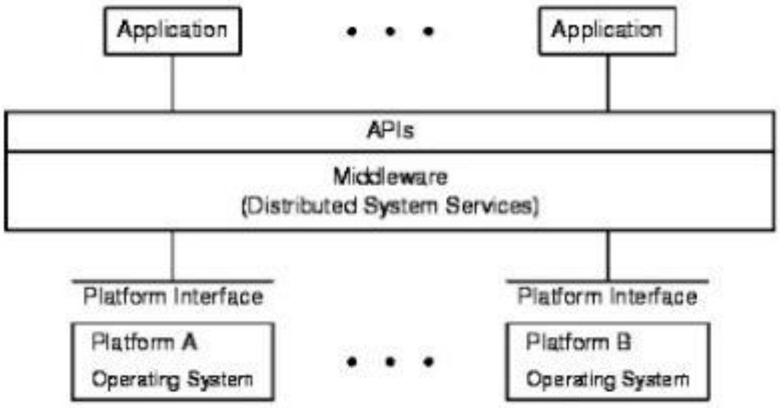


Figure 2. A distributed system organization as a middleware. The middleware layer extends over multiple machines, and offers each application the same interface. The figure is captured from [3]

Few examples of typical middleware services are:

Communication: Is well-known as RPC (Remote Procedure Call). A RPC service allows an application to invoke a function which is implemented and executed from a remote computer as if it was available locally.

Transactions: In this case, applications use multiple services which are distributed. Middleware generally offers special support which is called atomic transaction. In atomic transaction the application developer specifies the remote services and the middleware ensures that every service is invoked, or none at all by following a standardized protocol.

Service Compositions : This is notably the case for Web-based applications. In that case, web based middleware standardizes the way Web Services are accessed and provides the means to generate their function in a specific order. For example, mashups are web pages which combine and aggregate data from different sources.

<u>Reliability:</u> With specialized tools, like The Horus Toolkit [9], a developer is allowed to build reliable distributed applications. It can build, for example, an application as a group of process which guarantees that the message will be received by all. These applications are typically implemented as a part of middleware.

2.2 The ZeroMQ library

In this section there is description of the library which is used at the project to achieve message exchanging. At first is given an overview about the library. Then a reference to how the installation was done. After that a reference to request reply patterns which are available on ZMQ. At last is given a description of a C program with the use of ZMQ library which is done for the needs of the project and is explained the choice about the socket pattern

ØMQ

Figure 3. The ZeroMQ logo as it is appeared in [4]

2.2.1 What is ZeroMQ?

ZeroMQ (also known as ØMQ,0MQ) [4] is a library suitable for distributed messaging. This is the library which is used with C to achieve the message exchanging. It gives you sockets that carry messages across various transports like TCP, multicast. In addition you can connect N to N sockets using patterns like pub-sub, request-reply etc. ZeroMQ is developed by iMatix [8] and is LGPLv3 open source.

As you can see in Figure 4 there is the symbol Ø. The Ø on ØMQ is all about tradeoffs. From the one side makes the name invisible to Google and Twitter. On the other side some Danish people gets annoyed because they have similar letter “Ø” in their alphabet.

To be more specific some Danish people reacted on that and they sent a lot of messages including insults or bad words.

In reality the zero on ZeroMQ was meant as "zero broker" and "zero latency". As time was passing the zero get a lot of meanings like zero administration, zero cost, zero waste.

More generally “zero” means that the project has more power due to the fact that it removes complexity.

A a	B b	C c	D d	E e	F f	G g	H h
a	be	ce	de	e	ef	ge	hå
[æːˀ]	[b̥eːˀ]	[seːˀ]	[d̥eːˀ]	[eːˀ]	[ef]	[g̊eːˀ]	[hɔːˀ]
I i	J j	K k	L l	M m	N n	O o	P p
i	jåd	kå	el	em	en	o	pe
[iˀ]	[jʌð]	[kʰɔːˀ]	[el]	[em]	[en]	[oːˀ]	[pʰeːˀ]
Q q	R r	S s	T t	U u	V v	W w	X x
qu	ær	es	te	u	ve	dåbelve	eks
[kʰuːˀ]	[æɐ̯]	[es]	[tˢeːˀ]	[uːˀ]	[ʋeːˀ]	[dʌb̥əlʋeːˀ]	[eg̊s]
Y y	Z z	Æ æ	Ø ø	Å å			
y	set	æ	ø	å			
[yːˀ]	[sɛd̥]	[ɛːˀ]	[øːˀ]	[ɔːˀ]			

Figure 4. The Danish alphabet. Letter “ø” is at the bottom line

2.2.2 Sockets and patterns

ZeroMQ header files have a variety of functions which let you to handle sockets [4,5]. ZeroMQ sockets life is divided in four parts:

- Create and destroy them using functions zmq_socket() and zmq_close() in respectively
- Configure them by setting options on them and checking them if necessary using functions like zmq_setsocopt(), zmq_getsockopt()

- Plug them into network topology by creating ZMQ connections to and from them with functions zmq_bind()(on the one side), zmq_connect()(on the other side).
- Carry data by writing and receiving messages on them using functions zmq_msg_send(), zmq_msg_recv()

For some request reply patterns the message exchanging is done using methods s_send() and s_rcv() from zhelpers.h file. These two methods in their implementation they use zmq_msg_send(), zmq_msg_recv() respectively.

ZeroMQ socket patterns are implemented by pairs of sockets with matching types depending on what type of system you want to implement. Some legal patterns which ZeroMQ provides are:

- Requester to Replier
- Dealer to Router
- Publisher to Subscriber

Requester to Replier is the suitable pattern for programs where the replier sends a reply only when it receives a request from requester

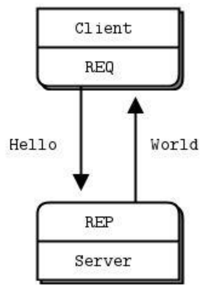


Figure 5. The requester replier socket pattern on client/server example captured from [4]

Dealer to Router is a more powerful socket combination. In that case, it gives the chance to nodes to send asynchronously request and replies. In other words, every node can send at any time whatever it wants without waiting for a reply. There is also a difference on message structure. In this pattern is used an empty delimiter frame. That happens because internally is used a request-reply pattern but applications cannot see

Publisher to Subscriber is a socket combination where a publisher can only send a message and subscriber can only receive a message. Subscriber cannot send back. Furthermore a publisher can set an identifier on the message which it will send. Subscriber from the other side will receive only the message with the specific identifier from that publisher.

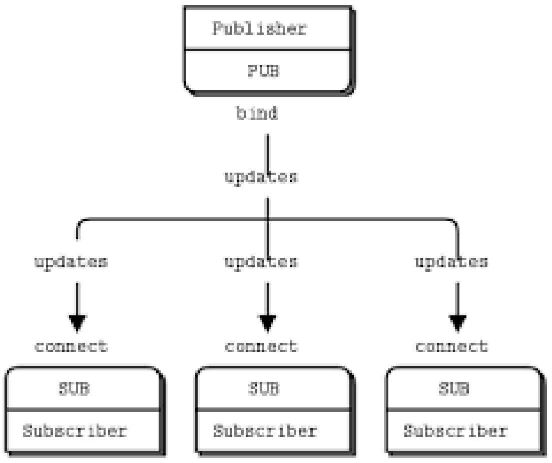
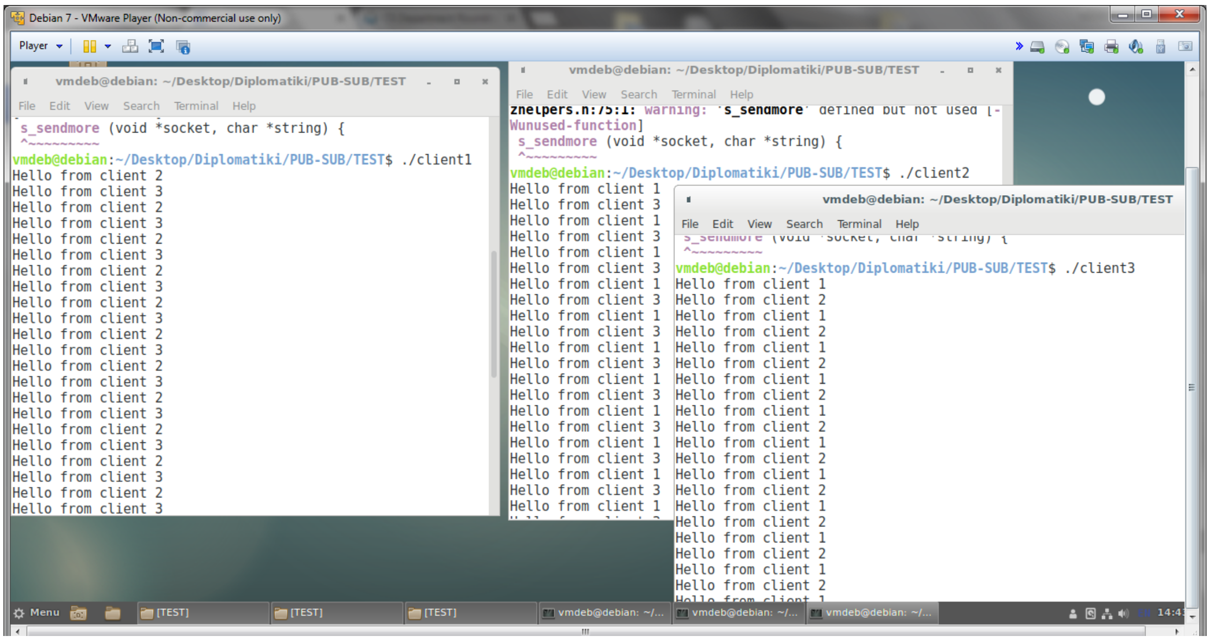


Figure 6. The publisher-subscriber socket combination captured from [4]

2.2.4 Example of a C program with the use of ZeroMQ library

Assume that we have three clients A, B and C. All clients send hello message to the others and receive hello message from the others too. To achieve this I selected the Dealer to Router combination because is a powerful socket combination where all clients can send asynchronously messages to each other. Each client has two channel ports and is connected on two sockets to receive message from the others. As about its sending sockets it opens a socket for each client to sends its messages and the others connect with it. In case, where we have two clients communicating, this has less complexity because each client has one channel port and is connected to one socket to receive messages. In Figure 7 we see a part of the execution of the program. Each terminal is for each client. As we see each client sends its hello message to the others and receives hello message from the other two.


Debian 7 - VMware Player (Non-commercial use only)

vmdeb@debian: ~/Desktop/Diplomatiki/PUB-SUB/TEST

```
 s_sendmore (void *socket, char *string) {
 ^~~~~~~~~~
vmdeb@debian:~/Desktop/Diplomatiki/PUB-SUB/TEST$ ./client1
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
Hello from client 2
Hello from client 3
```

vmdeb@debian: ~/Desktop/Diplomatiki/PUB-SUB/TEST

```
zhelpers.h:75:1: warning: 's_sendmore' defined but not used [-
Wunused-function]
 s_sendmore (void *socket, char *string) {
 ^~~~~~~~~~
vmdeb@debian:~/Desktop/Diplomatiki/PUB-SUB/TEST$ ./client2
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
Hello from client 3
Hello from client 1
```

vmdeb@debian: ~/Desktop/Diplomatiki/PUB-SUB/TEST

```
 s_sendmore (void *socket, char *string) {
 ^~~~~~~~~~
vmdeb@debian:~/Desktop/Diplomatiki/PUB-SUB/TEST$ ./client3
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
Hello from client 2
Hello from client 1
```

Figure 7. Part of the execution of the program where three clients talk to each other

2.3 Self Stabilization in Distributed Systems

Stabilization was firstly introduced by Edsger W. Dijkstra in 1974 [6]. Self-stabilization of a system guarantees of automatic recovery from any transient failure without additional effort. In other words regardless of current system's state the system will converge to a legal state in a finite number of steps. This means that, it is better to design distributed systems that can be started at any state and converge to a legal state.
An example to explain better the definition of "Self stabilizing" is like a space shuttle as discussed by professor Shlomi Dolev in [7]. The space shuttle may experience a fault, such as power supply problem and will have to recover automatically. As a result the control system of the space shuttle is design to be self-stabilizing.

2.3.1 Definition

Definition 1. A system is the pair S =(C,$\rightarrow$), where C is a set of states of the system S and $\rightarrow$ is a binary transition relation on C. A computation of S is a non-empty sequence(c1,c2,….) such that for all i≥0: c_j ∈C and $c_i \rightarrow c_{i+1}$, c_i express a global state.

Locally each process can be in a different state. All local states of each process and the contents of every communication channel are concatenated to form the global state.

2.3.2 General properties of self-stabilizing algorithms

When the system converges to a subset of legal states, it gains the advantage of automatic recovery from any system failure, despite of the state the system had been moved into and the quantity of the data that had been corrupted. Therefore it is obviously required that the failure has to be transient which means that the fault has to last for a short period of time.

Convergence implies three important properties of self-stabilization:

- Embedded tolerance of arbitary transient failures
- Unneeded proper initialization of the algorithm
- Obvious adaptivity to dynamic changes of the system configuration

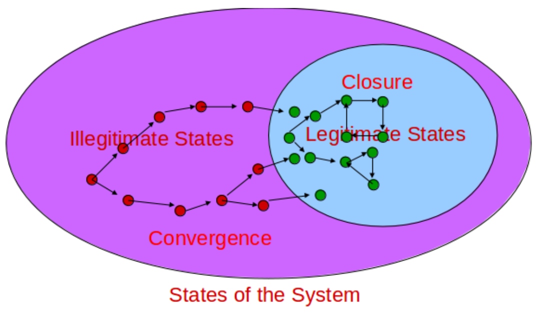


Figure 8. States of the system, legal or not and how the recovery is done

<u>Embedded tolerance of arbitary transient failures:</u> In case where a small part of the system is not working but the system is still working properly there will be no change to the state of each node.

<u>Unneeded proper initialization of the algorithm:</u> The initial state of the system it doesn't have to be legal. Self-stabilization acts whenever the system is in an illegal state.

Obvious adaptivity to dynamic changes of the system configuration: For example network topology changes (as any of them can be treated as transient fault).

On Figure 8 the closure describes the subset and the legitimate states are referenced to the legal states of the system. Each node has to be in one of these states. When all nodes are in one of these states we say that the system is on valid state. The states outside the closure are called illegitimate or illegal states. If some nodes are in these states then self-stabilization starts the convergence. Convergence in Figure 8, is described as the path of states which a node passes to recover from the transient fault. In other words, it follows a path where from the illegitimate states to be in legitimate (legal) states.

On the other hand there are some drawbacks of self-stabilizing algorithms like:

- Tolerance of only non-permanent faults
- Possible inconsistency of system states during convergence to legal states
- No explicit detection of accomplishing convergence

In addition during failure time, the system can behave arbitrarily. A self-stabilizing algorithm can be more complex and hard to construct.

2.4 Self-stabilization for two fundamental distributed computing problems

In this section they are described two fundamental distributed computing problems and how self-stabilization solves them. In particular, the two problems are mutual exclusion and leader election.

2.4.1 Problem 1: Mutual Exclusion

Mutual exclusion as it discussed in [6] is a well-known fundamental problem in distributed computing. Consider a distributed system with p processors. Each processor at any time may need to execute its critical section. Only one processor is allowed to use shared resource. The system must guarantee two properties:

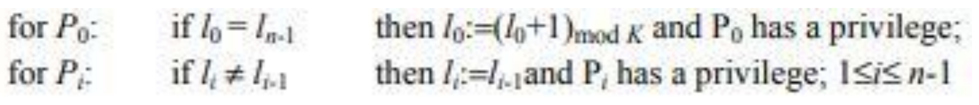
- Every time only one processor is allowed to access the critical section, not more (mutual exclusion)
- Every processor must be able to execute its critical section infinitely often (fairness)

On the other hand mutual exclusion can cause several problems to a system like deadlocks and mutual blocking, where no one processor is allowed to enter its critical section even if at least one processor is ready to enter in it.

This problem was firstly introduced by Edsger W. Dijkstra in 1974 [6]. It guarantees that whenever the system starts from an arbitrary state it will converges to a legal configuration in a finite number of steps. Self – stabilization algorithms can solve mutual exclusion but are not able to solve deadlock.

Assume that we have a model with n processors (P0, P2, P3,……., Pn-1) connected in a ring topology. This means that each processor has two neighbors one from the left side and one from the right side. For example for the Pi processor its left neighbor is Pi-1, except P0 where its left processor is Pn-1. As about the right neighbor for Pi processor, is Pi+1 except Pn-1 where its right processor is P0. In addition each processor has a variable Xi where it stores an integer value between 0 and n. For the purpose of the system there is a scheduler called central daemon which activates one processor at a time to execute a computational step. It is been assumed that this scheduler is fair and it chooses one process at a time.

Here in Figure 9 we can see the Dijkstra's algorithm



for P_0: if $l_0 = l_{n-1}$ then $l_0 := (l_0+1)_{\bmod K}$ and P_0 has a privilege;
for P_i: if $l_i \neq l_{i-1}$ then $l_i := l_{i-1}$ and P_i has a privilege; $1 \leq i \leq n-1$

Figure 9. Dijkstra 's self-stabilization algorithm captured from [6]

In Figure 10 we can see an exemplary run of this algorithm. In this example we have 3 processors connected as a ring, where K=3. The system has an initial state, all three processors enter in their critical section (P0 satisfies the first condition of the algorithm where $l_0 = l_{n-1}$, P1 and P2 satisfy the second condition where P1: $l_1 \neq l_0$ and P2: $l_2 \neq l_1$) and change their state in step 1. From mutual exclusion point of view this is not legal, because all of them enter its critical section and as a result we are in an illegal state. At second step we have the same situation where all processors enter their critical section again and so we have another illegal state. Now condition $l_0 = l_{n-1}$ is violated – process P0 cannot enter its critical section, but P1 and P2 can. As a result at step 3 only P1 and P2 change their state and so we still have an illegal state. Then after step 3, only P2 can enter its critical section. The system now is converged to a set of legal states. After that convergence only one processor will be allowed to enter its critical section. In other words, only one processor can use the shared resource at any time.

The algorithm needs O(n^2) system steps to reach a legal global state. The number of local states K depends on ring size n and must satisfy the relation: K≥n

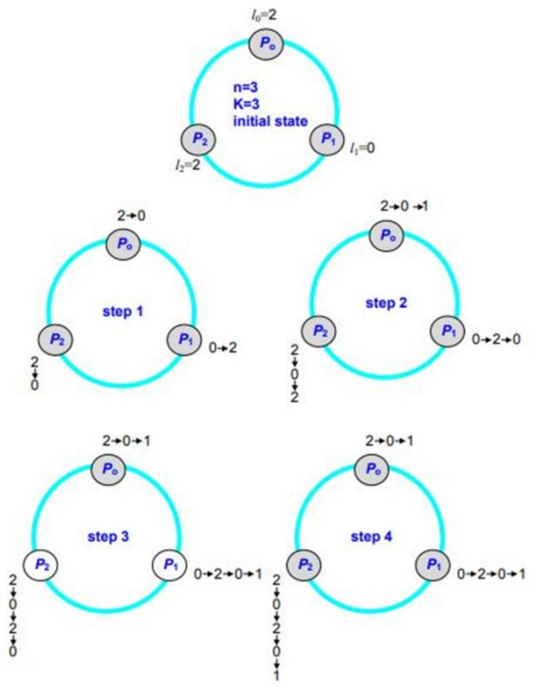


Figure 10 Exemplary run of Dijkstra’s stabilizing mutual exclusion algorithm captured from [6]

2.4.2 Problem 2: Leader Election

Leader election [6] is another fundamental problem in distributed system. In this problem we have a set of processors where we have to select one in order to perform a specific task. It is very necessary and useful to have a leader in the system especially in cases where a shared resource is a shared memory.

There are a lot of leader election algorithms which can solve this problem. However each one has a different time and communication complexity. Also they have differences, such as in terms of network topology, fault tolerance, the collapse and restore of processes in network etc. The major requirement which a leader election algorithm needs, is to inform all process in the system which process is elected as leader when the algorithm‘s execution terminates successfully. As about process identifiers in the algorithm they are unique identifiers but at initial state no one process know the identifiers of the other process.

This problem can be solved with self-stabilization too with the same guarantees, as the mutual exclusion problem which is discussed in previous subsection. Now we will describe a self-stabilization algorithm for the problem

The algorithm is randomized and solves the problem within an exponential expected number of rounds. In that situation, we have a model of complete graph systems in which each process communicate with all other processes via shared memory. Each process communicates with all other process using a single-writer, multi-reader, binary register, called *leader register*. *Leader_i* denotes the leader register of process Pi.

Idea of the algorithm: The algorithm solves the problem with floating identifier introducing coin toss. It uses a random algorithm which simulates a coin toss and return values 0 or 1. The algorithm starts with an arbitrary configuration with any possible combination of binary values of the leader registers. It fixes all leader registers by making all leader registers value 0 except one. After that the process which has stored in leader register the value 1 is elected as a leader in the system.

Chapter 3

Self-stabilizing Reconfiguration Scheme

3.1 Introduction

This chapter is an overview of self-stabilizing reconfiguration algorithm which I had to implement. The modules of the algorithm is described in [8], a technical report which I have studied. It was constructed by a group of four people which includes my supervisor Dr Chryssis Georgiou and Doctoral student Mr Ioannis Markoullis, from the University of Cyprus. The other two people of the team were Prof Shlomi Dolev from Ben-Gurion University of the Negev which is located at Beer-Sheva in Israel and Prof Elad M. Schiller from Chalmers University of Technology, located at Gothenburg in Sweden.

Nowadays there are a lot of configuration techniques which are based on starting the system in a consistent configuration in which all participating entities are in a predefined state. Many working systems cannot control two important things. Churn rate of processors which describes how many processors join and leave the configuration and access to unbounded storage. This automatically reconfiguration algorithm can recover from transient faults like those which were described above. Dynamic and difficult-to-predict nature of distributed systems is prone to transient faults especially due to hardware or software malfunctions and requires efficient solutions.

The algorithm considers a distributed system that works in dynamic asynchronous environment like a shared storage system. It can recover from transient faults like violations on predefined churn rates or the unexpected activities of processors and communication channels. Its self-stabilizing ability regains safety by assuming temporal access to reliable failure detector until safety is re-established in the system.

Configurations are a set of active processors which provide services of the system. As time passes a configuration may lose an amount of active participants if they voluntary leave or if they stop by failure. While the system works, new processors will come and ask to join. There is a need to allow new participations. New participants are used to form with remaining processors of the system a recent participation group.

3.2 System settings

The algorithm considers an asynchronous message passing system of processors. Each processor Pi has a unique identifier i, taken from totally ordered set of identifiers P. There is an upper bound N which declares that the number of live and connected processors at any time of the computation is N < |P|. Those processors are called active. They know the upper bound N which accounts for participants and those that are still joining. However they do not know how much active processors are there at any time.

Some processors may crush and stop working unexpectly. In that case a crushed processor never rejoins the computation. Rejoins, is a kind of transient fault and self-stabilization inherently deals with rejoins by regarding the past information as possibly corrupted.

Using the joining mechanism a new processor can join the system at any point in time with an identifier drawn from P. This identifier is used from a processor forever. When he joins the computation the processor becomes a participant but it is not a part of the configuration.

In addition each processor has a failure detector where it stores every other processor’s identifier when it receives a message. Processors which will be in the failure detector of

Pi, it assumes that those are alive in the system. As about how processors are ordered in a failure detector, the processor that has most recently contacted Pi is the first in Pi’s vector.

3.3 System Reconfiguration Task

A configuration is a bounded size set of processors which can be called as config. The system has a valid configuration (conflict-free), when no two processors, that are active in the system, have different configuration values. If there is a configuration conflict the system assigns the value $\perp$ (at the program is assigned the character ‘\0’). This process is called *configuration reset* and is a process for recovery from transient faults. This process ends, when the system will finally store valid configuration values to the processors of the system. In other words the process ends when the system state return to conflict-free.

When the system returns to a conflict free state, the participants can call for the establishment of new configurations. They propose a non-empty set of participants to replace current configuration. Processors which are newly arrived can become participants while no reconfiguration occurs. If there is a reconfiguration in progress, the system will not allow new participants to enter as well as it will not allow further reconfiguration requests. As a result, when there are no configuration conflicts, the system ability to replace the existing configuration with a proposed one depends on crush rate.

For this purpose, the system uses a failure detector. It stores all processors which a processor Pi assumes that is active. Failure detector has to be eventually and temporarily reliable in order to help system to attain a conflict free configuration. After that it can be unreliable until the crash rate allows it to be reliable in order to help the system again to replace current configuration with new one. If there is a violation which it has to do with failure detector, then a transient fault occurs and as a result the system shall recover through the configuration reset process.

3.4 The Scheme

The reconfiguration scheme is a composition of two layers. The reconfiguration stability assurance layer (recSA) and the reconfiguration management layer (recMA). The scheme is accompanied of the Joining Mechanism. In Figure 11 we can see the interaction between the modules and with the application. For the purpose of the project we emphasize on recSA and recMA layer and Joining Mechanism.

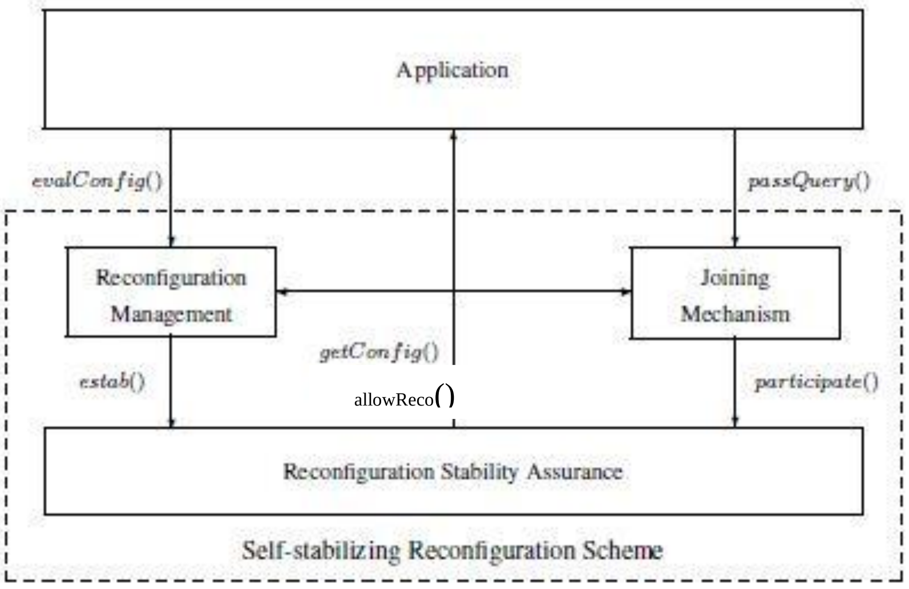

Application
evalConfig()
passQuery()
Reconfiguration Management
Joining Mechanism
getConfig()
estab()
allowReco()
participate()
Reconfiguration Stability Assurance
Self-stabilizing Reconfiguration Scheme

Figure 11. The self-stabilizing reconfiguration scheme, captured from [8]

The recSA layer ensures that all participants eventually have a common configuration set. It uses the interfaces getConfig() and allowReco() to introduce processors that want to join the computation and to provide information on current configuration and on whether a reconfiguration is not taking place. The reconfiguration management uses a prediction function evalConfig() to evaluate when a reconfiguration is required. It also checks that by monitoring if the configuration loses its majority.

If a reconfiguration is needed the recMA layer initiates it with estab(). Finally, Joining mechanism proceeds only when there is no reconfiguration in progress and there is a reconfiguration in place. Newly arrived processors use it to join as participants in the system. The direction of the arrows shows the transfer of a specific information from a module A to a module B.

Chapter 4

Algorithm Description and Implementation

4.1 Reconfiguration Stability Assurance

4.1.1 Description

Reconfiguration Stability Assurance layer referenced as recSA layer which is discussed in [8] is a self-stabilizing algorithm which ensures that all computing elements of the system have the correct configuration. The algorithm gets information from Reconfiguration management layer whether is necessary to have a reconfiguration process.

It guarantees the following three things:

- All processors have the same configuration
- When participants notify the system to satisfy their wish for reconfiguration, the system has to select one proposal to replace the current configuration
- Joining processors can become participants eventually

The algorithm follows the configuration replacement automaton as it is shown in Figure 12. When a processor is in phase 0, it means that the system works fine and we have to monitor for stale information.

At phase 1, is the point where the reconfiguration process starts. Each processor has a proposal for new configuration. Before phase 2 processors are deciding which configuration is better. Usually the selected proposal is the one where most of processors have in common with the others.

When a processor is in phase 2 it means that it is remained only one proposal which replaces the existent one. Then the automaton returns back to phase 0 and the system starts the monitoring of stale information.

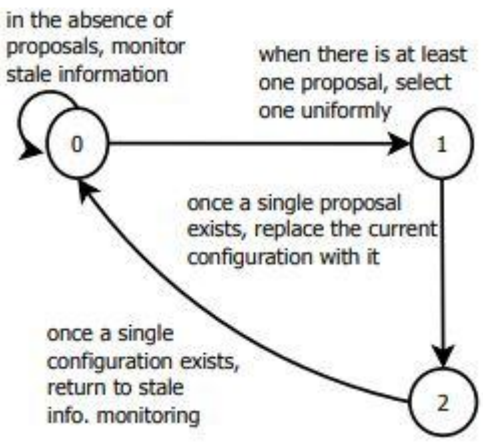

in the absence of proposals, monitor stale information
when there is at least one proposal, select one uniformly
0
1
once a single proposal exists, replace the current configuration with it
once a single configuration exists, return to stale info. monitoring
2

Figure 12. The configuration replacement automaton, captured from [8]

Furthermore the algorithm combines two techniques:

Brute force stabilization: The system recovers from stale information which is a type of transient fault. For example, a configuration conflict where two or more processors have different configurations. Then Pi assigns ⊥ to its config value and then starts the reset process where it nullifies all config fields of the system. After that it assigns at its config field the set of the active processors which are stored in the FDi I, where FDi is Pi’s failure detector. When this process ends, processors which were joined at the system will finally become participant.

Delicate (configuration) replacement: This process is used when processors move to phase 1 where is the time when they start to propose sets for configuration replacement. Participants can propose the current configuration with a new one set through the estab(set) interface.

Configuration replacement process, when it starts after any call to the estab(set) interface, controls the replacement using the following three phases:

- Selects uniformly and deterministically a single proposal
- Replaces all config fields with the selected proposal
- Brings the system back to state 0 (according to automaton) where the system starts monitoring for stale information

Each processor stores in a table, let's say prp [ ] the phase which is a processor at any time and its proposal set which is non-empty when a processor's phase is 1 or 2 and $\perp$ when processor is at phase 0. The last referred proposal $< 0, \perp >$ is called default proposal. In that case, a function of the algorithm which is called allowReco( ) returns false which means that any processor can join the system. In cases where phase of the processor is 1 or 2 the allowReco( ) function returns true which means configuration replacement is in progress.

Every time when a processor changes its phase it informs the others at the end. However, no one processor starts its new phase until all the other participants come to the same phase with it. In other words according to the algorithm it stores a value in a table called all [] where all [i] becomes 1 when Pi sees that all other participants hold the same notification. This means that all processors are in the same phase. Also Pi has an allseen list where it stores all processors from which it receives the all indication.

Using this method when a processor reaches at phase 2 it means that it has a single proposal set which at the end is the same with the other participants set when all the other participants reaches phase 2 to. At the end, the algorithm choose the proposal set and replace the configuration and finally all participants, new and old, return to phase 0 and the monitoring for stale information starts again.
The whole procedure above executes forever using an infinite loop and only processors which its view of the configuration is different from # - denotes that a processor is a participant.

4.1.2 Implementation

At first I had to implement a data structure where each processor has to be able to store the necessary information about other processors. It has to know at any time, some details which are referenced below:

- Which processors are in the current configuration
- Which processors are alive/trusted
- Which processors are active
- In which phase is each algorithm and if it is at phases 1 or 2 which set are having as proposal
- If all trusted processors had noticed its current notification and holds the same
- Which processors of the configuration know about the phase of a Pi processor
- Which is the most recent value which a processor Pi received from a processor Pj

To be more specific let's see what variables does the algorithm using a pseudo code as they are implemented:

Config[]: This array contains all processors view on current configuration. In position i (config[i]) there is Pi's view on current configuration.

FD[]: This array represents Pi's failure detector where Pi stores ides of processors which are trusted. The processor which sends last to Pi is first at Pi's failure detector. This for Pi means that this processor is 100% trusted.

FD_part[]: This array is for active participants. FD_part[i] is an alias for $\{Pj \in FD[i] : config[j] \neq \# \}$.

Prp[]: In this array are stored the configuration replacement notifications. These notifications are in pairs < phase, set>. One field is for the phase of the processor and the other one is for the proposal set. In other words, in prp[i] there are Pi's most recent phase, where phase ∈ {0,1,2} and proposal set , where set $\subseteq$ P, for new configuration which Pi proposes.

All[] : An array where is stored the boolean value *all indication*. It indicates that a processor Pi observes that all trusted processors had noticed its current (maximal) notification and they hold the same notification.

allSeen: A list which include all processors which they received the all[k] indication

echo[] : An array in which echo[i] is (FD_part[i], prp[i], all[i])'s alias and echo[j] refers to the most recent value that Pi received from Pj after Pj had responded to Pi with the most recent values it got from Pi.

The above variables are inside a structure which is called PROCESSOR. The size for all arrays is a constant variable called MAX_C which represents the maximum number of processors which can join the system. Also in each position of the array it is stored the identifier of the processor to make searching more easy. Furthermore each array has its own structure depending on what kind of values it stores

The structure which holds the variables for the reconfiguration stability assurance algorithm looks like this:

```
Struct Processor{
   Config[MAX_C];
   FD[MAX_C];
   FD_part[MAX_C];
   Prp[MAX_C];
   All[MAX_C];
   Echo[MAX_C];
   List *allseen;
}PROCESSOR;
```

In addition echo’s array type comes from another structure which called ECHO and it looks like this:

```
Struct Echo{
   FD_part_echo[ ] ;
   Prp_echo[ ] ;
   All_echo[ ] ;
}ECHO;
```

4.2 Reconfiguration Management

4.2.1 Description

This layer which is called recMA, is responsible to decide when a reconfiguration is necessary. It takes this decision when:

- The configuration loose its majority
- The prediction function called evalConf() finds out that more than the 1/4th of the trusted processors are not working in configuration.

Using the estab(set) interface the recMA layer informs the recSA layer that a reconfiguration has to take place when one of the two situations occurs. Several processors may trigger reconfiguration simultaneously but each processor triggers a reconfiguration only one time when it needs it. The proposed set from each processor for the configuration contains processors which are trusted participants. This algorithm is executed only from a processor which is an active participant.

4.2.2 Implementation

According to self-stabilizing reconfiguration scheme from Figure 11, the reconfiguration management module uses the interface evalConf() to evaluate if the reconfiguration is required. This is achieved using a simple prediction function where it returns True when the 1/4th of the configuration members are not trusted (processors which are not included in Pi’s FD) by a processor Pi, otherwise it returns false.

In addition this module uses the interfaces allowReco() and estab(set) from reconfiguration stability assurance module. The first one returns True when a reconfiguration is required or false if it is not. In that case, this means that the system works fine without stale information inside and new processors can join the computation. The second interface initiates the creation of a new reconfiguration. If one of the reconfiguration case, which the module checks, occurs then each Pi stores in its prp set its FD_part aggregate (active participants) and moves to phase 1 according to automaton which is shown in Figure 12. Finally from recSA module it uses the getConfig() function which returns the agreed configuration or ⊥ if reconfiguration process is taking place. Due to the fact that the module uses interfaces from recSA at the implantation the recMA module includes the recSA’s library.

As about the variables which the module needs, they are two integers which work as flags which each processor uses. The first one is called needReconf and the second one is called noMaj. Both variables take only the values 0 and 1. When the needReconf value is 1 this means that a processor Pi says that a reconfiguration is required otherwise the value is 0. On the other hand when noMaj value is 1, this denotes that in configuration members there is no majority and if the value is 0 it means that the majority exists. Both variables for implementation reasons are part in the structure which is used at reconfiguration stability assurance algorithm. Each processor has its own table called flags [ ] where inside it stores its values and the other processor values of needReconf and noMaj when it receives them during message exchanging

Furthermore the algorithm uses the macros Core( ) and flushFlags( ). Macro Core( ) collects common processors which are assumed as active from processors in the configuration. FlushFlags( ) macro is used from each processor Pi where it sets needReconf and noMaj flag of each processor Pj, which is in its FD, to 0.

4.3 Joining Mechanism

4.3.1 Description

Joining mechanism is used only from processors which want to join the system. They use a snap stabilizing protocol to avoid introducing stale information after it establishes a connection with the system processors. The joiner enters in an infinite loop sending joining requests to each processor which is at its FD aggregate. To be allowed a processor to join the configuration from participants it depends on what is happening to the system. If there is a reconfiguration in progress then a processor waits to join. When there is no reconfiguration in progress the config members decides if it is okay to accept a processor who wants to join. Each processor answers back to joiner by sending it the pass value and state which represents its local variables. Finally, when more than half of config members have positive decision and processor agrees with application terms, it joins the configuration.

4.3.2 Implentation

The mechanism uses the interfaces allowReco( ), participate( ) and getConfig( ) from recS. The allowReco( ) as it descripted above returns True if reconfiguration is required and False if the system works fine. Participate( ) makes Pi participant by assign it the aggregate of configuration members at its config [ ] in i-th position. GetConfig( ) returns the agreed configuration or ⊥ if reconfiguration process is taking place.

In addition it uses the following variables. FD[ ] as defined in recSA algorithm, state[ ] and pass[]. State is a table where each processor Pi stores Pj 's local variables values. Due to the fact that the application does not exists, for implementation reasons dummy values for state are used. Pass [ ] is also a table where each processor Pi stores pass value for each processor Pj which is a configuration member. If the value is 1 it means that processor Pj accept a processor's joining request and if the value is 0 it means that processor Pj rejects a processor's joining request.

Functions resetVars( ) and initVars( ) for implantation reasons are not implemented. As a result when is the time to be used at an execution, a printing message appears which says which function of them is used.

Chapter 5

An implementation example

5.1 Scenario: Accepting a joiner to become a participant

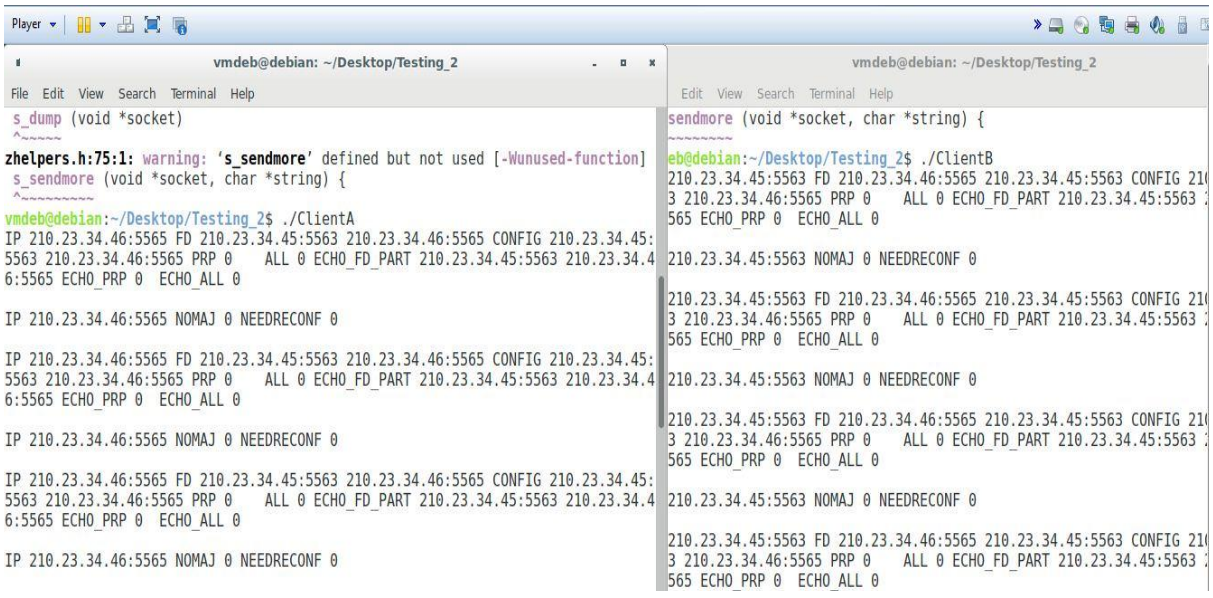
Assume that we have two processors in the configuration. Let's say A and B. Those processors send messages to each other executing the recSA and recMA modules of the algorithm at the same time. The system is at phase 0 where, according to the automaton of the recSA module, the system works fine and at any time a new processor can join the system. Later a processor C uses the joining mechanism and asks to join. Processors A and B accepts and finally processor C becomes a participant.

5.2 Preparation

At first the libraries from the three modules of the algorithm were combined together in a single library to work properly as it is described in Chapter 4. The program which is described in Chapter 3 was the basis for the implementation of the simulation where three processors are connected with each other and can exchange messages. Then the library included to each C file with the appropriate libraries which a C file needs to execute by default. To make the simulation work properly each processor had fixed details in its tables. For the needs of the scenario processors which were in the configuration, had details like configuration view, all processors included was part to each other processor's FD, all processor were in phase 0 with no proposal set, all flags initialized to 0 e.t.c. As about the part of each algorithm, which each processor is executed, it was combined in a single infinite loop where the algorithm of recSA module was executed before the algorithm recMA module. The joining mechanism executed by processor A and B when processor C sent joining requests

5.3 Simulation of the example

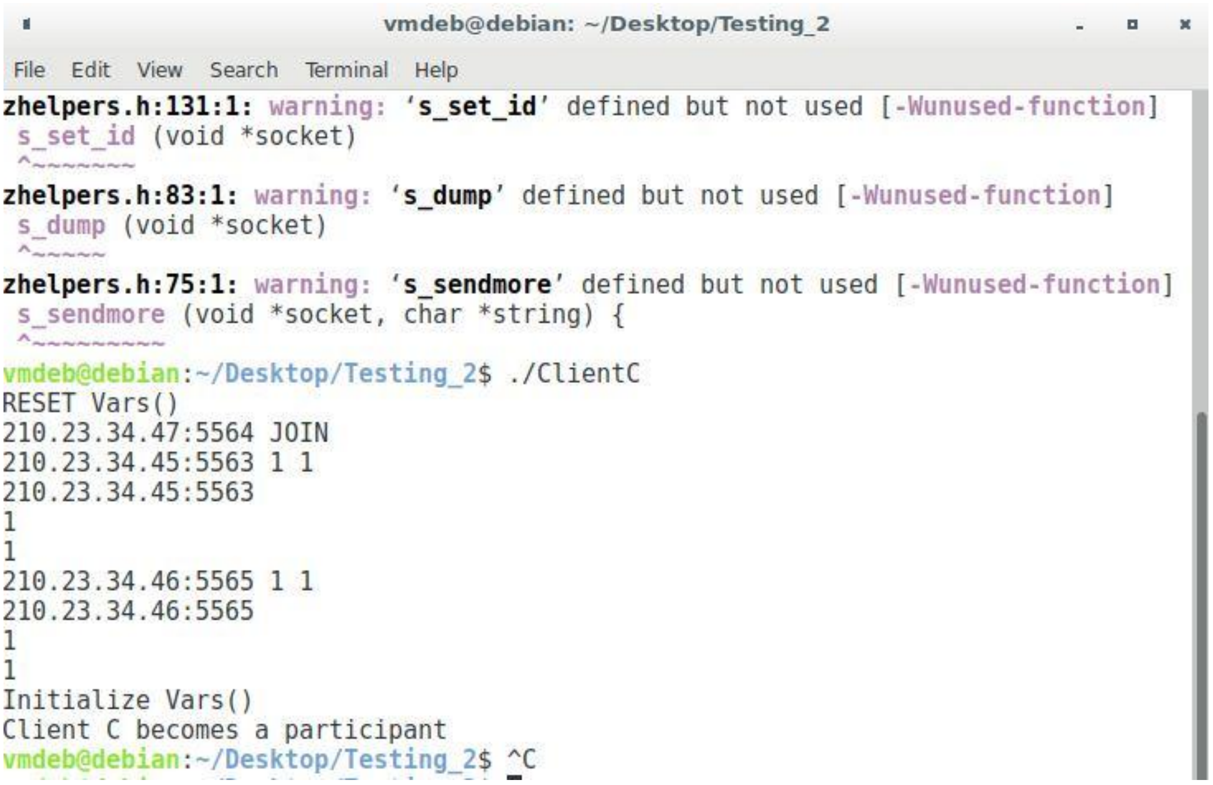
As it is shown in Figure 13 at first part of the execution processors A and B send messages to each other. After some time processor C executes its joining mechanism to start send joining requests to processor A and B as it is shown in Figure 14. Immediately processors A and B respond to C accepting its request because there was no reconfiguration process taking place. Then processor C joins the system as a participant while processors A and B continue their communication like before. According to the correctness of the joining modules is proven that configuration members accepts new processor to join only if no reconfiguration process taking place.



```
vmdeb@debian: ~/Desktop/Testing_2
 s_dump (void *socket)
 ^~~~~~
zhelpers.h:75:1: warning: ‘s_sendmore’ defined but not used [-Wunused-function]
 s_sendmore (void *socket, char *string) {
 ^~~~~~~~~~
vmdeb@debian:~/Desktop/Testing_2$ ./ClientA
IP 210.23.34.46:5565 FD 210.23.34.45:5563 210.23.34.46:5565 CONFIG 210.23.34.45:
5563 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563 210.23.34.4
6:5565 ECHO_PRP 0  ECHO_ALL 0

IP 210.23.34.46:5565 NOMAJ 0 NEEDRECONF 0

IP 210.23.34.46:5565 FD 210.23.34.45:5563 210.23.34.46:5565 CONFIG 210.23.34.45:
5563 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563 210.23.34.4
6:5565 ECHO_PRP 0  ECHO_ALL 0

IP 210.23.34.46:5565 NOMAJ 0 NEEDRECONF 0

IP 210.23.34.46:5565 FD 210.23.34.45:5563 210.23.34.46:5565 CONFIG 210.23.34.45:
5563 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563 210.23.34.4
6:5565 ECHO_PRP 0  ECHO_ALL 0

IP 210.23.34.46:5565 NOMAJ 0 NEEDRECONF 0
```

```
vmdeb@debian: ~/Desktop/Testing_2
sendmore (void *socket, char *string) {
~~~~~~~~~
eb@debian:~/Desktop/Testing_2$ ./ClientB
210.23.34.45:5563 FD 210.23.34.46:5565 210.23.34.45:5563 CONFIG 21
3 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563
565 ECHO_PRP 0  ECHO_ALL 0

210.23.34.45:5563 NOMAJ 0 NEEDRECONF 0

210.23.34.45:5563 FD 210.23.34.46:5565 210.23.34.45:5563 CONFIG 21
3 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563
565 ECHO_PRP 0  ECHO_ALL 0

210.23.34.45:5563 NOMAJ 0 NEEDRECONF 0

210.23.34.45:5563 FD 210.23.34.46:5565 210.23.34.45:5563 CONFIG 21
3 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563
565 ECHO_PRP 0  ECHO_ALL 0

210.23.34.45:5563 NOMAJ 0 NEEDRECONF 0

210.23.34.45:5563 FD 210.23.34.46:5565 210.23.34.45:5563 CONFIG 21
3 210.23.34.46:5565 PRP 0    ALL 0 ECHO_FD_PART 210.23.34.45:5563
565 ECHO_PRP 0  ECHO_ALL 0
```

Figure 13. Exemplary run of Client A and Client B



```
vmdeb@debian: ~/Desktop/Testing_2
zhelpers.h:131:1: warning: ‘s_set_id’ defined but not used [-Wunused-function]
 s_set_id (void *socket)
 ^~~~~~~~
zhelpers.h:83:1: warning: ‘s_dump’ defined but not used [-Wunused-function]
 s_dump (void *socket)
 ^~~~~~
zhelpers.h:75:1: warning: ‘s_sendmore’ defined but not used [-Wunused-function]
 s_sendmore (void *socket, char *string) {
 ^~~~~~~~~~
vmdeb@debian:~/Desktop/Testing_2$ ./ClientC
RESET Vars()
210.23.34.47:5564 JOIN
210.23.34.45:5563 1 1
210.23.34.45:5563
1
1
210.23.34.46:5565 1 1
210.23.34.46:5565
1
1
Initialize Vars()
Client C becomes a participant
vmdeb@debian:~/Desktop/Testing_2$ ^C
```

Figure 14. Exemplary run of Client C

Chapter 6

Conclusion

6.1 General Conclusions

Self-stabilization is a very useful ability at distributed systems. It helps a system to converge from an illegal state to a legal in finite number of steps and without an additional human effort. This is very important because in a lot of distributed systems their computing elements are spread in different geographically places around the world. This makes it more difficult for someone to monitor all elements of the system for unexpected behavior.

An unexpected behavior of an element is a very common problem for a distributed system which works in an asynchronous environment. Modules which are implemented according to their description which is described in [8] can solve these kind of problems.

As about ZeroMQ library, nowadays with its communication services can solve a lot of problems which are appeared in a distributed system. ZeroMQ has the ability to construct communication sockets to send information to other over TCP and UDP. In addition it gives you a lot of examples and tutorials to build your own distributed system as you wish and it has plenty of function in its libraries which can be very helpful. Also it gives the chance to a programmer to use it in different programming languages and not only in C where ZeroMQ is originally implemented

6.2 Problems and Assumptions

The first problem that I had to face was with the installation of the library. At first I tried to install it on windows but the library was not recognized on minGW compiler and on Cygwin. Then I try to install it on Ubuntu with no effect. Finally I managed to install it on Debian which I used it through a virtual machine. In addition due to the fact that my laptop has installed windows 7 professional 32 bit I install an oldest version of Virtual Machine which is called VMware.

In addition if there were runtime errors depending on ZeroMQ functions the compiler was not able to discover the error. In that case the program during the execution was stacked. An example is when I was trying to send messages through anif statement without putting the algorithm in a loop and also if I was trying to read messages but the receiver buffer was empty.

Another problem which I faced with the library was with the character “\0” which was representing that the field has no information. The problem was that at the receiver side this character was not existed. For each message, which a processor sent through my program was separated with a whitespace. If the message included the “\0”, the receiver‘s side when it used the string tokenizer function(strtok()) the “\0” character was identified as a whitespace too. As a result some information had stored differently causing problems to the algorithm to work properly.

6.3 Future Work

First of all it is better for someone who is going to use my program to execute it on a physical machine with Ubuntu or Debian installed. This will help him to achieve the communication between two PCs using the IP address.

All the functions from the three modules are implemented correctly but the general structure of the program to allow an amount of nodes to use the algorithm was not correct. It must ask user to enter number of nodes which wants to execute the program and from file each node takes an identifier and according to the scenario the program must be able to make the appropriate socket connections.

Furthermore it must be checked the performance of brute force stabilization process and delicate configuration. In other words it must be checked how the system reacts after a transient fault occurs and how much time it wants to make the reconfiguration process depending on the number of nodes which are part of the system.

6.4 Benefits from the thesis

Working on this subject I had the chance to explore distributed system domain of computer science and learn new things which I did not have the chance to do it through my years at university. I had the chance to learn a lot of things about distributed systems and self- stabilization ability. In addition I learn how to use the ZeroMQ library which was new to me and I got more experience with the use of C programming language.

Despite of the difficulties that I have to accomplish the whole project I think the knowledge that I gain is very good. Maybe in the near future this knowledge can be useful. For example if I have to build algorithms which work on distributed system.

Finally I learned how to cope with the methodology of research where you have to find everything yourself. Furthermore at any time when you develop a program or a system requirements will be change without warnings and you have to be able to make changes to your code to satisfy them.

Bibliography

[1] Maarten van Steen, Andrew S. Tanenbaum, A brief introduction to distributed systems, University of Twente, Enschede, The Netherlands, 2016

[2] Jerzy Brzezinski, Michal Szychowiak, Self-stabilization in distributed systems - a short survey , Institude of Computing Science, Poznan University of Technology

[3] Rishikese MR, Middleware concept and middleware in distributed applications, 2014,[Online]. Available: https://www.slideshare.net/Rishikese/middleware-and-middleware-in-distributed-application

[4] ZMQ guide, [Online]. Available: http://zguide.zeromq.org/page:all

[5] Pieter Hintjens, ZeroMQ Messaging for Many Applications, O'Reilly Media, Inc.,
1005 Gravenstein Highway North, Sebastopol, CA 95472, [Online]. Available: http://www.it-ebooks.info

[6] Christina Georgiou, Despina Vacana, Marita Thoma, Natalie Temene, Self – Stabilization for Three Fundamental Distributed Computing Problems, Department of Computer Science, University of Cyprus, Cyprus

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

[8] Shlomi Dolev, Chryssis Georgiou, Ioannis Markoullis, Elad M. Schiller, Self-Stabilizing Reconfiguration, In the Proceedings of NETYS 2017

[9] van Renesse R, Birman K, Cooper R, Glade B, Stephenson P (1994) The horous system. In: Birman K, van Renesse R (eds) Reliable and distributed computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos, pp 133-147