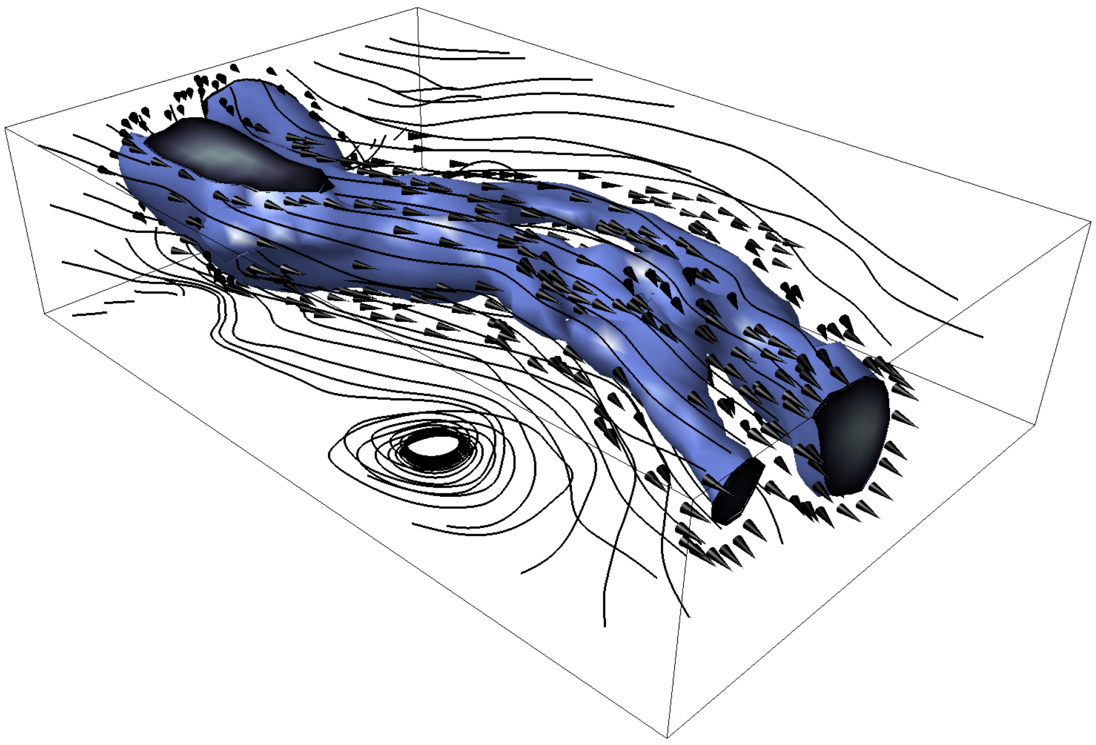
CHALMERS
UNIVERSITY OF TECHNOLOGY



Implementation and Assessment of a Self-Stabilizing, End-to-End Communication Algorithm in TinyOS

Master's Project in Computer Science & Engineering

Evert Boelaert

Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

MASTER'S PROJECT 2016

Implementation and Assessment of a Self-Stabilizing, End-to-End Communication Algorithm in TinyOS

Evert Boelaert

Department of Computer Science & Engineering
Division of Networks & Distributed Systems
Distributed Computing & Systems Research Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Implementation and Assessment of a Self-Stabilizing,
End-to-End Communication Algorithm in TinyOS

Supervisor & examiner: Elad Michael Schiller,
Department of Computer Science and Engineering,
Networks and Distributed Systems Division

Master's Project 2016
Department of Computer Science & Engineering
Division of Networks & Distributed Systems
Distributed Computing & Systems Research Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

End-to-end communication is an essential part of any communication network. The main goal is the need for reliable, stable, and fast file transfer in a distributed network.

Sensor networks have grown in size and popularity over the past decades. The need for a reliable way of collecting the data that these sensors measure, has grown with it. Unreliable communication channels often hold a risk for corrupting the data that the sensors collect. That is why we keep looking for new ways to improve the communication channel.

At the Chalmers University of Technology in Gothenburg, Sweden, researchers have thought of an algorithm that could potentially solve these problems of bad communication in unreliable networks. They provide a self-stabilizing, end-to-end communication algorithm in bounded capacity, omitting, duplicating, and non-FIFO dynamic networks.

We do our research towards finding any effort or literature that is working towards the practical implementation of such an algorithm. Next, we look into the background of distributed networking and wireless sensor networks. We explain the thought process behind the algorithm and the theoretical inner workings.

Before we start the practical implementation, we look at the challenges ahead and formulate the central research question.

In the methods section, we provide a detailed look into the development process of the algorithm. We give an overview of the tools needed to complete this work, and the obstacles that had to be overcome.

The results provide the reader with the proof that the algorithm works in a practical implementation. We execute a number of experiments in different scenarios, which consist of single-fault errors or combinations of these errors that are introduced in the communication according to a predefined probability.

Finally we provide the reader with a short overview of what can be done with these results, and we propose a direction in which future research can go.

Keywords: self-stabilization, end-to-end, communication, TinyOS, FIFO, bounded capacity, nesC.

Acknowledgements

As a Belgian exchange student, given the opportunity to work on such a project at one of the best schools in Sweden, I need to thank a few people who made this possible and who have supported me during the work.

First and foremost I'd like to thank my supervisor at Chalmers University of Technology, Elad Michael Schiller, for giving me the fantastic opportunity to work on this project, for providing me with all the tools I needed, and for guiding me along the way. Secondly I'd like to thank Olaf Landsiedel for helping me in times of need, when I needed help, tips, and guidance during development.

I also want to show my appreciation for my Swedish friends in the Master Thesis workroom at Chalmers. Daniel Moreau, David Alm, Robin Karlsson and Henning Phan. They were great colleagues from the start, and they have become friends.

Last but certainly not least, I wish to show my utmost appreciation to all the persons at my home intsitution, the University of Leuven at the Technology Campus in Geel, who helped me a great deal to be able to spend the past semester in Sweden. They even kindly provided their assistance during my stay here, whenever I needed their advice. These persons include, but are not limited to; Peter Karsmakers, Patrick Colleman, Hilde Lauwereys, Geert Van Ham, Patricia Van Genechten and Isabelle Moons.

Evert Boelaert, Gothenburg, January 2016

Contents

1 Introduction

End-to-end communication is in many ways the basics of communication networks. Sending a message from one point, the sender, to another, the receiver. It is what we call a basic primitive in communication networks. The information must be able to arrive at the receiver, one part of the whole message at at a time. No omissions, duplications or reordering of the data are allowed.
Because of outside noise and interference, errors tend to unfold in the network between the communicating entities. It is especially at moments when the communication network experiences high loads, that the chance of errors becomes very high, and eventually we can't guarantee stable communication anymore.
In this category of problems, Wireless Sensor Networks (WSNs) are one of the technologies that are inherently the most prone to interference[10]. The deployment of WSNs has been steadily on the rise in recent years, particularly due to the availability of sensors that keep getting smarter, cheaper, and more intelligent due to big advances in sensor technologies. But WSNs and sensors in general are often deployed in outside environments and very harsh conditions, like volcanoes[15]. It's no surprise then that these little devices are quite prone to catching noise.
A major goal of the research in this field has been to remove this outside noise and interference out of the equation as much as possible, or minimize the effects of it on the network, so we are able to guarantee reliable and well performing end-to-end communication[2]. Removing the interference is not always easy to accomplish, since we often have little control over the outside world. But we can, however, target the errors in the communication channel that this interference causes. Thus, error detection and error correction are two fields where research is most focused, in the hopes of minimizing errors as much as possible. In combination with these technologies, research looks toward the concept of self-stabilization. This means that an algorithm should be able to recover from any arbitrary state, after encountering an error for example.
Chalmers University, and specifically my supervisor, Elad Michael Schiller, have developed an algorithm that employs these two main technologies: error correction codes and self-stabilization. The algorithm they present can be applied to dynamic networks of bounded capacity that omit, duplicate and reorder packets.

1.1 Background and related work

Gulliver[13], is a platform for studying vehicular systems on a large scale open source test-bed of low cost miniature vehicles that use wireless communication and

are equipped with onboard sensors. It was developed at Chalmers University of Technology, and my supervisor was part of the team that presented it. The paper about self-stabilizing end-to-end communication that is the building block of this thesis, was written as part of a series of papers presented as part of the Gulliver project.
At the time of presenting the algorithm, this was the only algorithm of its kind. There many attempts in the general direction, but none could satisfy all the guarantees that this algorithm can[4]. My assignment is to implement the algorithm and make practical proof regarding the theoretical statements the original paper makes in terms of guaranteeing stable communication.
After doing thorough background work, I have come to the conclusion that there are still no efforts or algorithms produced that can guarantee the same kind of stable communication. At the time of writing this report, the best efforts concerning this type of application are still the ones referred to in the original paper, so I will not repeat them here. There is also, to the best of my knowledge, no practical implementation of any kind available.

1.2 My contribution

This project investigates the practical implementation in TinyOS and presents the working proof of the presented self-stabilization algorithm. This project presents the first, to the best of my knowledge, practical implementation and proof of a self-stabilizing end-to-end algorithm for reliable FIFO message delivery over bounded non-FIFO and duplicating channel.

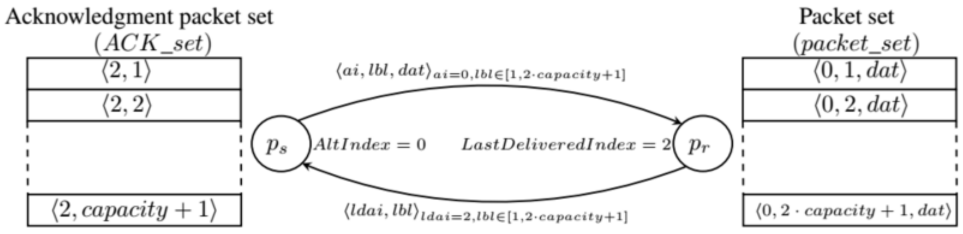
2
Theory

The theoretical algorithm is divided in two parts, in accordance with the way the algorithm is build up in the original paper[4], and maps on to the actual development cycle I went through when implementing the algorithm.

1. We start with the first attempt algorithm. This forms the foundation for the advanced version.
2. The advanced, full S^2E^2C (**S**elf-**S**tabilizing, **E**nd-to-**E**nd **C**ommunication) algorithm consists of the expanded first attempt algorithm, together with a packet formation algorithm based on Error Correction Codes (ECC).

2.1 First attempt algorithm

We start with this first algorithm before moving on to the advanced version. It is a self-stabilizing, large overhead, end-to-end communication algorithm for coping with packet omissions, duplications, and reordering. We have two sides, a sender algorithm and a receiver algorithm.



Acknowledgment packet set
(ACK_set)
$\langle 2,1\rangle$
$\langle 2,2\rangle$
$\langle 2, capacity+1\rangle$
$\langle ai, lbl, dat\rangle_{ai=0, lbl\in[1,2\cdot capacity+1]}$
p_s
$AltIndex = 0$
$LastDeliveredIndex = 2$
p_r
$\langle ldai, lbl\rangle_{ldai=2, lbl\in[1,2\cdot capacity+1]}$
Packet set
($packet\_set$)
$\langle 0,1,dat\rangle$
$\langle 0,2,dat\rangle$
$\langle 0, 2\cdot capacity+1, dat\rangle$

Figure 2.1: The first attempt algorithm.

The sender starts by fetching a batch of messages from the application layer. Each data packet is put in a message of the form **<Ai; lbl; dat>**, where *dat* represents the data, or the actual message packet. The sender then sends the packet to the receiver. The two other variables are the Alternating Index, and a label. The label increments until *<CAPACITY+1>* every time a new message is send. The Alternating Index will only increment in modulo 3 when a new message batch is fetched. This means that for every value of the *Ai*, *<CAPACITY+1>* labels map on to it.
The receiver gets the message delivered, reads the values and puts it in a position in a *packet_set* array. The position in the array is not randomly chosen, but depends on the *label* value. The Receiver then sends an *ACK* (acknowledgement) message back

to the sender. When the sender receives this message, it knows the current message was delivered at the destination and it puts the incoming acknowledgement in the **ACK_set** array. The sender keeps the Alternating Index the same, but increments the label value. During this process where the receiver gets the message, puts it away, and sends an acknowledgement message, the sender keeps transmitting the same message. It only stops when the acknowledgement arrives. This is the principle of the **Alternating Bit Protocol** or ABP[1], and it is just slightly different from the well known Stop-and-Wait ARQ protocol[5], which waits with sending new messages until an acknowledgement arrives.
Once the receiver has received the full batch of messages and the incoming label reaches *<CAPACITY+1>*, the receiver sets it's **Last Delivered Alternating Index** value to the incoming *Ai*, and delivers the messages in *packet_set* to the application layer on the receiver side.
At this point both sides reset their arrays and the cycle starts again with new messages.

2.2 Advanced: packet formation from messages

To add an extra layer to the algorithm, the **packet_set()** function is added. It takes a batch of messages of length *pl* and size *ml* (per message), regards this batch of messages as a 2D bit-matrix, and transposes this whole matrix. Instead of a **pl x ml** size matrix we now have a matrix with **n** amount of rows, where $n > ml$ because of the added redundancy bits from the Error Correction Coding. Figure 2.2 shows a graphical representation of this process.

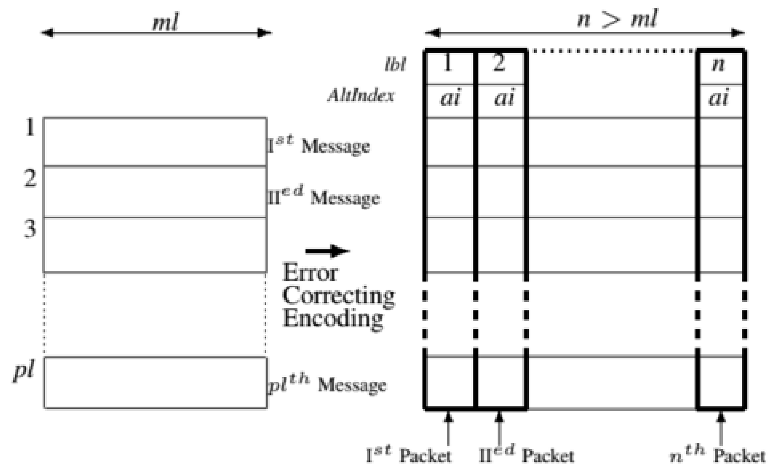


Figure 2.2: Packet formation from messages.

By combining the first-attempt algorithm with the advanced step of adding error correction codes, we become the full S^2E^2C algorithm.
Figure 2.3 shows the complete sender algorithm, and figure 2.4 shows the complete receiver algorithm. A more thorough theoretical explanation can be found in the paper "Self-Stabilizing End-to-End Communication in Bounded Capacity, Omitting, Duplicating and non-FIFO Dynamic Networks"[4].

```
Algorithm 1: Self-Stabilizing End-to-End Algorithm (Sender $p_s$)
Local variables:
$AltIndex \in [0,2]$ : state the current alternating index value
$ACK\_set$: at most $(capacity + 1)$ acknowledgment set, where items contain labels and last delivered alternating indexes, $\langle ldai, lbl \rangle$

Interface:
$Fetch(NumOfMessages)$ Fetches $NumOfMessages$ messages from the application and returns them in an array of size $NumOfMessages$ according to their original order
$Encode(Messages[])$ receives an array of messages of length $ml$ each, $M$, and returns a message array of identical size $M'$, where message $M'[i]$ is the encoded original $M[i]$, the final length of the returned $M'[i]$ is $n$ and the code can bare $capacity$ mistakes

Function $packet\_set()$ begin
    foreach $(i,j) \in [1,n] \times [1,pl]$ do let $data[i].bit[j] = messages[j].bit[i]$
    return $\{\langle AltIndex, i, data[i] \rangle\}_{i \in [1,n]}$
Do forever begin
    if $(\{AltIndex\} \times [1, capacity + 1]) \subseteq ACK\_set$ then
        $(AltIndex, ACK\_set, messages) \leftarrow ((AltIndex + 1) \bmod 3, \emptyset, Encode(Fetch(pl)))$
    foreach $pckt \in packet\_set()$ do send $pckt$
Upon receiving $ACK = \langle ldai, lbl \rangle$ begin
    if $ldai = AltIndex \wedge lbl \in [1, capacity + 1]$ then
        $ACK\_set \leftarrow ACK\_set \cup \{ACK\}$
```

Figure 2.3: The full sender algorithm, expressed in pseudo code.

```
Algorithm 2: Self-Stabilizing End-to-End Algorithm (Receiver $p_r$)
Local variables:
$LastDeliveredIndex \in [0,2]$: the alternating index value of the last delivered packets
$packet\_set$: packets, $\langle ai, lbl, dat \rangle$, received, where $lbl \in [1,n]$ and $dat$ is data of size $pl$ bits

Interface:
$Decode(Messages[])$ receives an array of encoded messages, $M'$, of length $n$ each, and returns an array of decoded messages of length $ml$, $M$, where $M[i]$ is the decoded $M'[i]$. The code is the same error correction coded by the sender and can correct up to $capacity$ mistakes
$Deliver(messages[])$ receives an array of messages and delivers them to the application by the order in the array

Macros:
$index(ind) = \{\langle ind, *, * \rangle \in packet\_set\}$

Do forever begin
    if $\{\langle ai, lbl \rangle : \langle ai, lbl, * \rangle \in packet\_set\} \not\subseteq \{[0,2] \setminus \{LastDeliveredIndex\}\} \times [1,n] \times \{*\} \vee$
    $(\exists \langle ai, lbl, dat \rangle \in packet\_set : \langle ai, lbl, * \rangle \in packet\_set \setminus \{\langle ai, lbl, dat \rangle\}) \vee$
    $(\exists pckt = \langle *, *, data \rangle \in packet\_set : |pckt.data| \neq pl) \vee 1 < |\{ AltIndex : n \leq |\{$
    $\langle AltIndex, *, * \rangle \in packet\_set\}|\}|$ then $packet\_set \leftarrow \emptyset$
    if $\exists\, !\ ind : ind \neq LastDeliveredIndex \wedge n \leq |index(ind)|$ then
        foreach $(i,j) \in [1,pl] \times [1,n]$ do
            let $messages[i].bit[j] = data.bit[i] : \langle ind, j, data \rangle \in index(ind)$
        $(packet\_set, LastDeliveredIndex) \leftarrow (\emptyset, ind)$
        $Deliver(Decode(messages))$
    foreach $i \in [1, capacity + 1]$ do send $\langle LastDeliveredIndex, i \rangle$
Upon receiving $pckt = \langle ai, lbl, dat \rangle$ begin
    if $\langle ai, lbl, * \rangle \notin packet\_set \wedge \langle ai, lbl \rangle \in (\{[0,2] \setminus \{LastDeliveredIndex\}\} \times [1,n]) \wedge |dat| = pl$ then
        $packet\_set \leftarrow packet\_set \cup \{pckt\}$
```

Figure 2.4: The full receiver algorithm, expressed in pseudo code.

3 Methods

My method section is divided in three main parts. The **tools** section gives an overview of the most important tools I used for this project. Next, the **implementation** details the different phases of development. It gives insight in some of the coding techniques I used. Finally, the **testing & automation** section shows the development process I went through to automate the experiments. I went from testing the algorithm, to inserting errors, gathering the right data, and eventually automating this whole process.

Figure 3.1 gives a visual representation of the development structure I described above.

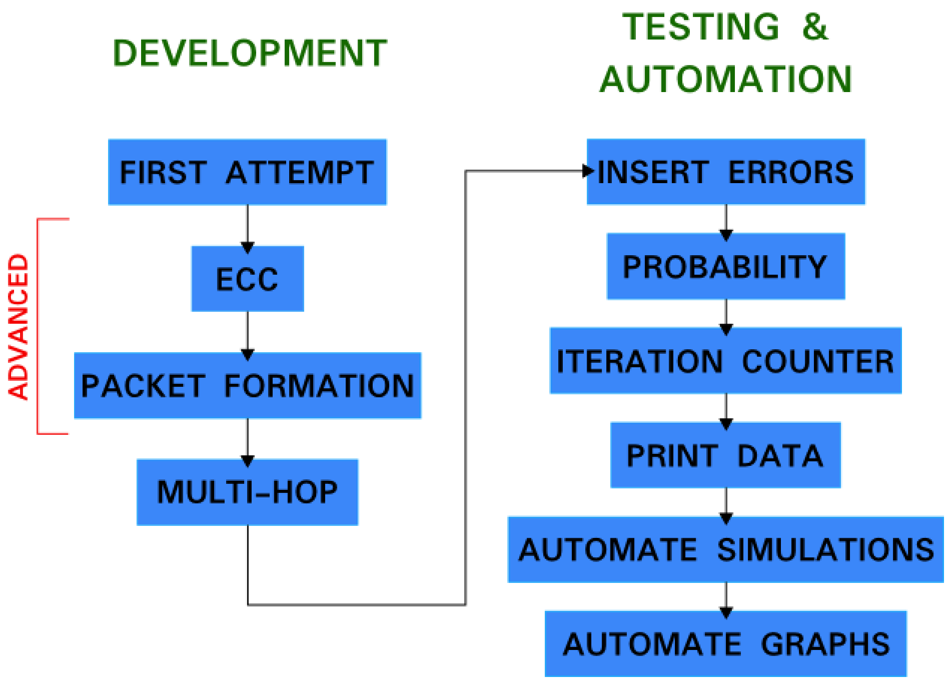


Figure 3.1: The figure shows my complete development structure I used to implement, and subsequently test and automate the algorithm. The arrows indicate the chronological sequence in which I came to the end product. On the left is the pure development of the algorithm, on the right starts the testing and automation part.

3.1 Tools

In this part I'll discuss the tools I used to implement and evaluate the S^2E^2C algorithm. For a full explanation of the details of each tool, I refer to my list of references. Instead, I will focus here on the elements of each tool that I specifically applied in my work.

3.1.1 LibReplay

LibReplay is a debugging tool, developed by Olaf Landsiedel, Elad Michael Schiller and Salvatore Tomaselli at Chalmers University of Technology. It is specifically designed to solve the problem that bugs in sensor networks present: they are very hard to hunt down because of the following reasons[11].

- Sensor networks are inherently based on distributed networking principles. This means that they operate in a non-deterministic environment.
- As a result of this first problem, bugs are time-consuming to track and reproduce.
- Bugs usually arise because of a very particular, complex sequence of events.

In sequential programming bugs are hunted down by integrated debugging capabilities that are found in Integrated Development Environments. These kind of debugging capabilities such as stepping through code, breakpoints, and watchpoints would significantly ease the debugging process. However, we are incapable of pausing program execution on a node because of the distributed and embedded nature of sensor networks.

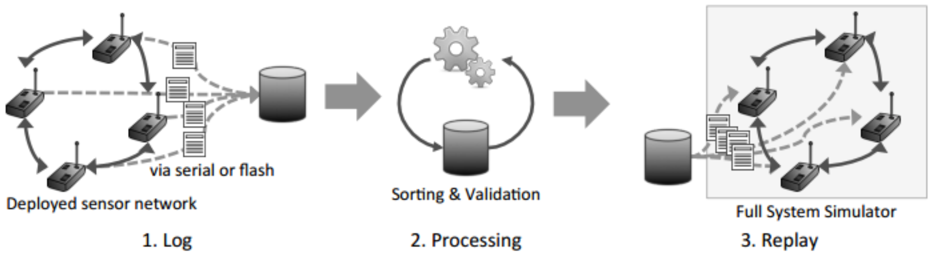


Figure 3.2: The above figure shows the way LibReplay works in two (three) steps: logging, processing the data, and replaying the exact state by feeding the algorithm all execution variables and necessary information to recreate the bug.

LibReplay solves these issues by logging function calls to and from the application or code of interest. Afterwards it is possible to replay the execution step-by-step in a controlled environment such as a full-system simulator like Cooja.

I will use LibReplay as a means to run experiments that are not possible in the normal mode of operation of sensor networks. These experiments include ommitting, duplicating and reordering packets on the fly.

3.1.2 Github

The most used Version Control System (VCS) for distributed revision control and source code management (SCM) on the Internet is Github[3]. It is based on Git. It provides a flexible and fast Linux client as well as a web-based graphical interface. I have used it for the whole duration of my work for a safe place to store the code, and revert back to working code in case something went wrong. Therefore all the source code for my thesis will be made available at `https://github.com/AlkylHalide/SEC_Algorithm`.

3.2 Implementation of the algorithm

I divided the development of the algorithm in four phases. I started the code from scratch, each new version building upon the previous one and adding functionality. The name of each version is in accordance with the conventions as described in the original paper for my assignment[4].

- First Attempt
- Advanced algorithm with Error Correction Encoding
- Full algorithm (packet formation from messages)
- Full algorithm with multi-hop routing

In the following sections I will highlight and explain parts of the code for each version.

3.2.1 First attempt algorithm

This is the first attempt version of the algorithm as described in the original paper[4]. The Sender sends each message with an Alternating Index value and a unique Label for each message. The Receiver acknowledges each message upon arrival and the Sender will only send the next message if the current one has been properly acknowledged. Once the Receiver receives a certain amount of messages, the algorithm delivers these messages to the Application Layer. The variable <*capacity*> determines this amount of messages in the network, the value of which can be adjusted according to the needs of the implementation in the algorithm itself.

3.2.1.1 Header file *SECSend.h*

I use the application header file to define two very important elements. These are crucial to the operation of the algorithm.
The first is the **AM Type** of each message. In this case this is `AM_SECMSG` and `AM_ACKMSG`. *AM* stands for Active Message. In the next section I'll explain the usefulness of this.

```
AM_SECMSG = 5,
AM_ACKMSG = 10,
```

The second very important element in the header file is the definition of custom message structure. TinyOS possesses a standard message format for sending messages over the network, namely *message_t*. This message format allows for custom structures which you can use to define your own message fields. This is perfect for the algorithm, since we have to send message-specific info with each packet along the network like the alternating index, a label, the data and the node id.
I have defined two message structures according to the requirements of the algorithm;

1. One structure *SECMsg* with four fields: alternating index, label, data, node id. It is used to transmit the data messages.
2. The second structure, *ACKMsg*, is used for the acknowledgement messages from the receiver. It houses only three fields: last delivered alternating index, label, and node id.

```
typedef nx_struct SECMsg {
  nx_uint16_t ai;
  nx_uint16_t lbl;
  nx_uint16_t dat;
  nx_uint16_t nodeid;
} SECMsg;

typedef nx_struct ACKMsg {
    nx_uint16_t ldai;
    nx_uint16_t lbl;
    nx_uint16_t nodeid;
} ACKMsg;
```

An important detail to note here, are the **nx** prefixes before all variables. In nesC, this prefix is used to define *external types*. These are an extension to C, and allow definition of types with a platform-independent representation. They are intended for communication with entities external to the nesC program (e.g., other devices via a network), hence their name [6].

3.2.1.2 Component file *SECSendC.nc*

nesC uses two main types of files: module files and configuration files. Modules consist of the actual application code, while configurations are used to assemble other components together. The interfaces are implemented in the module file, while the connection of the interfaces with the respective components are provided by the component file. This is called the **wiring** of interfaces and components. A nesC application can be uniformly described by a configuration file that wires together the components[6].
I've explained earlier that one of the difficulties of working with TinyOS and nesC, is the component linking model. The component file, ending with a **C** in accordance with the TinyOS naming conventions, links the components and interfaces together that form the nesC application at compile time.
Two lines are especially important, and they refer back to the *AM Types* that I defined in the header file. These are also the only lines in the component file that

are different between the sender and the receiver application.

Sender

```
components new AMSenderC(AM_SECMSG);
components new AMReceiverC(AM_ACKMSG);
```

Receiver

```
components new AMSenderC(AM_ACKMSG);
components new AMReceiverC(AM_SECMSG);
```

It's not uncommon to have multiple services using the same radio to communicate messages across the network. Naturally, these services can't connect all at the same time.To solve this problem, TinyOS provides the Active Message (AM) layer as a way to multiplex the access to the communication channel. Using this **AM type** to initialize the sender and receiver components, we can easily filter the messages that arrive at the receiver channel.

Upon inspection you'll notice the sender and receiver components have reverse *AM* types assigned to their *AMSenderC* and *AMReceiverC* component. Expanding upon the explanation of these AM types, this is logical. The sender sends data messages with a *SECMsg* structure, so the *AMSenderC* component is initialized with this AM type. Likewise, it receives acknowledgement messages from the receiver with a *ACKMsg* structure, so it is initialized with the *AM_ACKMSG* AM type. The receiver follows the same logic, but in the reverse direction.

3.2.1.3 Module file ***SECSendP.nc***

The module file, ending with a **P** again in accordance with the TinyOS naming conventions, is the module file. This is where we write the actual logic of the algorithm. I will expand upon the specific logic I've implemented, in the coming sections.

3.2.2 Error correction codes

This is the first attempt algorithm, together with the the error correcting codes.

When implementing error correction codes, there are a few options to choose from. The most commonly found codes used in implementations of sensor networks and more generally low-power and resource-constrained devices, are Reed-Solomon codes. They offer high performance and work reasonably well in this sort of communication because they can make up for errors of longer length than other Error Correcting Codes[16][14].

The disadvantage of working with Reed-Solomon is that they're quite difficult to implement correctly. To save time on my development cycle, I've used an external library for Reed-Solomon error correction. It provides me with a C implementation which I added to my own algorithm [7].

3.2.3 Packet formation from messages

These are the full Sender and Receiver algorithms as described in the paper. On top of the basic first attempt functionality the algorithm adds Error Correction Codes, divides the messages in packets and sends them to the Receiver.
When the messages are fetched from the application layer at the sender, these messages are divided into packets according to a special transformation. We regard the array of messages in bit format as a two dimensional array. We transpose this whole array, and then divide the result in even packets.
To obtain this result I've defined a function called `packet_set()`. This is the sequence I make the messages go through: I fetch the messages from the application layer, convert each message to its appropriate bitwise representation, the result is put in a two dimensional matrix, transposed, and converted back to integers to generate the packets. These packets are then passed on to the sending function. Figure 3.3 provides a visual representation of the process that takes place.

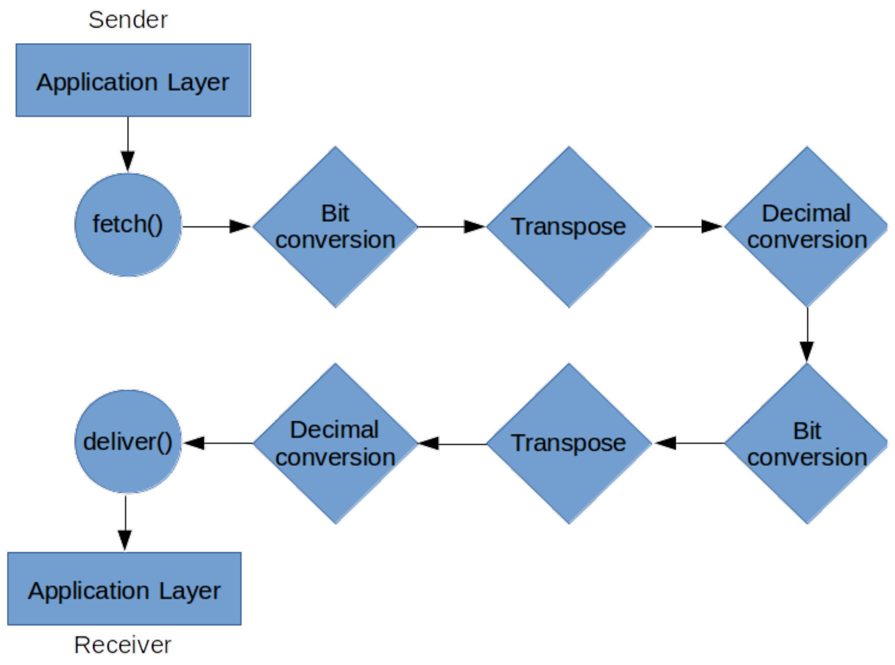


Figure 3.3: This figure shows the packet_set() function which is used to generate packets. It is clear to see that the sender and receiver functions are each other's opposites.

3.2.4 Multi-hop routing algorithm

In a real-life situation, it is very likely that the Sender and Receiver nodes are not within radio distance of each other. To fully observe the performance of the algorithm in such a real-life environment, it is therefore necessary to add the functionality of Multi-Hop routing through an appropriate Multi-Hop algorithm. In this case we have chosen for the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL, pronounced 'Ripple'). Unfortunately this brings along significant changes in the code to the Sending and Receiving algorithm. The reason is that this protocol uses UDP functionality to send and receives messages. To do this it uses custom TinyOS UDP functions as defined in BLIP 2.0 (Berkeley Low-Power IP Stack).

As we can see, the send and receive functions are now UDP-specific. This also means that we lose the ability to give AM types to outgoing messages. More importantly, it means we can't check incoming messages for their AM type. Luckily the 6LowPan implementation in TinyOs provides an AM field in the frame format, so I switched to incorporate that in combination with RPL[8].

UDP receive function

```
event void RPLUDP.recvfrom(struct sockaddr_in6 *from,
void *payload, uint16_t len, struct ip6_metadata *meta){
```

UDP send function

```
call RPLUDP.sendto(&dest, &btrMsg, sizeof(nx_struct SECMsg));
```

The use of 6LowPan and RPL routing means we switch to IPv6 addressing for sending to the other nodes. In point-to-point network, this requires that the address of the receiving node is known at compile town. Again I found a nice implementation of this feature already. The general prefix for IPv6 addressing is set in the Makefile on the following line.

```
PFLAGS += -DIN6_PREFIX=\"fec0::\"
```

This means that all the nodes will have an IPv6 address starting with `fec0::\`. The last part of the IPv6 address is used to refer to the specific node address. In the case of TinyOS, the last part can be directly filled in with the receiving node's node id. This makes it easy for us, since we were already addressing nodes this way.

```
memset(&dest, 0, sizeof(struct sockaddr_in6));
dest.sin6_addr.s6_addr16[0] = htons(0xfec0);
dest.sin6_addr.s6_addr[15] = (TOS_NODE_ID + SENDNODES);
```

3.3 Testing & automation

This entails the final phase in my development process. To test the algorithm I insert errors in a single-fault or mixed fault setup. I handle the automation by creating a series of BASH scripts that execute the right command-line commands in the Linux terminal. Figure 3.1 already gave a complete visual representation of the development process, I reiterate the sequence here shortly.

1. Insert errors in the communication.
2. Add a probability factor, which dictates the chance for an error to occur.
3. Implement a simple iteration counter.
4. Print meaningful data in the right way to the terminal, and make sure the data is saved in a text file.
5. Automate the simulations: this means to automatically go through all probabilities, one by one, for all experiments.
6. Automate the generation of the right graphs from the data.

3.3.1 Inserting errors

As we discussed earlier, we insert three type of errors in the communication between sender and receiver.

- **Omission**: a package is dropped or deleted entirely, in between the transport from sender to receiver.
- **Reordering**: we assume non-FIFO communication. This means packets can arrive at the receiver in a completely different order than how they were sent out.
- **Duplication**: it is possible duplicate packets arrive at the receiver, with a certain time interval between them. If this happens, the receiver should take the latest information (last packet to arrive) and overwrite the old packet with it in its packet set.

3.3.2 Probability

The rate at which errors are inserted in the communication, is determined by a probability factor. This is a number between 0, meaning zero percent, and 100, meaning hundred percent. It is obvious that at these extremes, communication is either completely error-free, or the communication between sender and receiver completely blocks because there are only errors generated.
For each experiment, we choose six probabilities in the rang of zero to hundred. We do this in a way that makes sure we create interesting experiments. This means not choosing the probability too low, or too high.

3.3.3 Iteration counter

The iteration counter is a simple variable that keeps track of the current amount of iterations of the algorithm. An iteration is said to be complete if it starts in the loop's first line, and ends at the last, regardless of whether it enters branches[4].

3.3.4 Output data (print)

As I described earlier in the part on LibReplay, the best solution for small scale debugging in TinyOS and nesC, is using the `printf()` command. The output gets sent to the serial port of the mote. Implementing this functionality is straightforward, and a tutorial on using the TinyOS Printf library can be found on the wiki page[17].
The difficult part however, is reading the information from the serial port of the mote. As said earlier, I use the full system simulator Cooja to simulate and test the algorithm. Remember that Cooja is part of Contiki, and so there is no direct support for connecting to the virtual mote in the simulator. The solution for this is opening a command-line interface and using the built-in TinyOS Java serial *PrintfClient* application. This application is able to take an argument through which you can specify along which communication channel the client should connect. In total it requires three steps to set up this communication channel.

1. Open a Cooja simulation and load an image in a mote.
2. Right-click on the mote and select *Mote tools → Serial socket (SERVER)*. The serial server socket plugin will open. Note the serial port, you can change this if you want.
3. Open a command-line interface and execute the following command. When you start the simulation, the `printf()` output will appear here.

```
    java net.tinyos.tools.PrintfClient
    -comm network@127.0.0.1:SERIAL_PORT
```

3.3.5 Automating simulations

To completely automate the simulations so they can run in the background, without supervision, requires the use of **Contiki test scripts**. This is a standard plugin in Cooja. Using this test script, I wrote a number of Linux Shell Scripts to capture the data being transmitted by the nodes.
A particular difficulty is to initiate the simulation in command-line mode (no GUI), and, at the same time, initiate the listener script. To start them parallel from each other, I used the parallel processing capabilities of shell scripts. The code is seen below.

```
. /home/evert/tinyos-main/apps/SEC/coojasim.sh &
. /home/evert/tinyos-main/apps/SEC/serial_connect.sh &
wait
echo "Processes complete"
```

In the code above I call two other scripts: *coojasim.sh* and *serial_connect.sh*. The first one starts Cooja in **nogui** mode, and it automatically loads the preferred simulation that you want to run. The second script connects to the serial interface of the mote(s) in Cooja, which I already discussed in an earlier paragraph of this methods section.

The trick of the script lies in the parallel processing. Behind every script is the *&* symbol, and the second to last line has the keyword *wait*. This means the script will call both scripts and let them run in the background.
This code is obviously a small, but important part of a much bigger shell automation script. In the complete automation script I loop through all my experiments, using a different error probability value each time.

3.3.6 Automating graph generation

To build the graphs I used the standard Linux OS tool for this function, GNUPlot[9]. I automated this by listing the GNUPlot commands in a shell script, and subsequently using this script as an input to GNUPlot. An example of such a script is shown here.

```
#!/bin/bash

reset

set boxwidth 0.5
set style fill solid
set xlabel "Probability"
set ylabel "Iterations"
set title "Omitting packets"
set yrange [0:1100000]
set key off
set style line 1 lc rgb "blue"
set style line 2 lc rgb "red"
set term png
set output "/home/evert/tinyos-main/apps/SEC/Graphs/
omitting_advanced.png"
plot "/home/evert/tinyos-main/apps/SEC/Graphs/
omitting_advanced.dat" every ::0::0 using 1:3:xtic(2)
with boxes ls 1, "/home/evert/tinyos-main/apps/SEC/
Graphs/omitting_advanced.dat" every ::1::2 using
1:3:xtic(2) with boxes ls 2
replot
```

4

Experiments & results

The goal of these experiments is to show how the different versions of the algorithm perform under the load of single fault errors, and errors that are a combinations of different faults.
To reiterate, the possible faults we can come across are omission, duplication, and reordering of packets during communication.
We differentiate between the first attempt algorithm with a large overhead, and the advanced, full S^2E^2C algorithm that uses Error Correcting Codes for a better efficiency.
We assess the performance of both algorithms based on the amount of iterations it goes through in a certain time limit. This time limit is an hour, and is equal for all experiments. The results shown are the outcome of the iteration variable for each chosen probability. We choose a total of seven error probability values for each experiment. Using the same probabilities gives us a good reference point for comparing the two versions of the algorithm in each circumstance.
The first probability bar graph in each experiment is marked in red, and is the result of using a probability value of zero. This is the error-free performance of the algorithm and is used as a reference point for the other results in the graph.

4.1 Single-fault error cases

For the single fault error cases, we subject only the first attempt algorithm to each experiment.

4.1.1 Omission

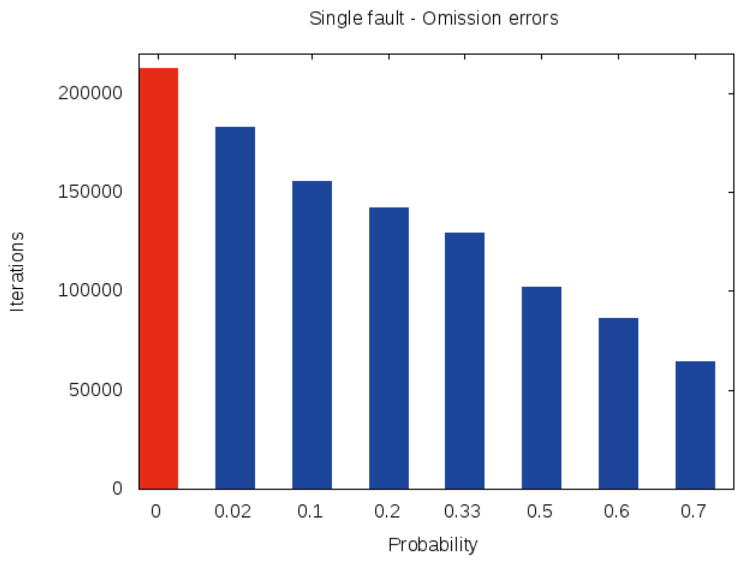


Figure 4.1: The figure shows the results of the first experiment. We test the first attempt algorithm for all error probability values, when only omission errors occur. We notice the quite constant downhill trend toward the higher probability values. This is a logic result for the first attempt version, because of the high overhead. At the two highest error rates, 0.6 and 0.7, the performance dips below 50 percent. The performance at the third highest error rate, 0.5, just makes this 50 percent barrier.

4.1.2 Duplication

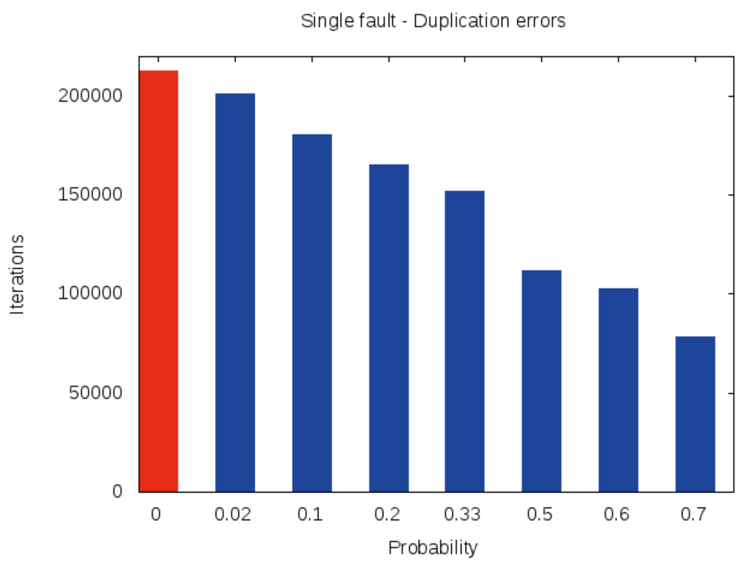


Figure 4.2: The figure shows the results of the second experiment. We test the first attempt algorithm for all error probability values, when only duplication errors occur. We notice the trend from the first experiment returning. The two highest probability values have a performance below 50 percent in comparison to the benchmark, again. The performance in the first half of the graph (probability values 0.33 and lower) is better compared to the omission experiment though. This is likely because of the smaller overhead required to correct duplication errors, compared to omission errors.

4.1.3 Reordering

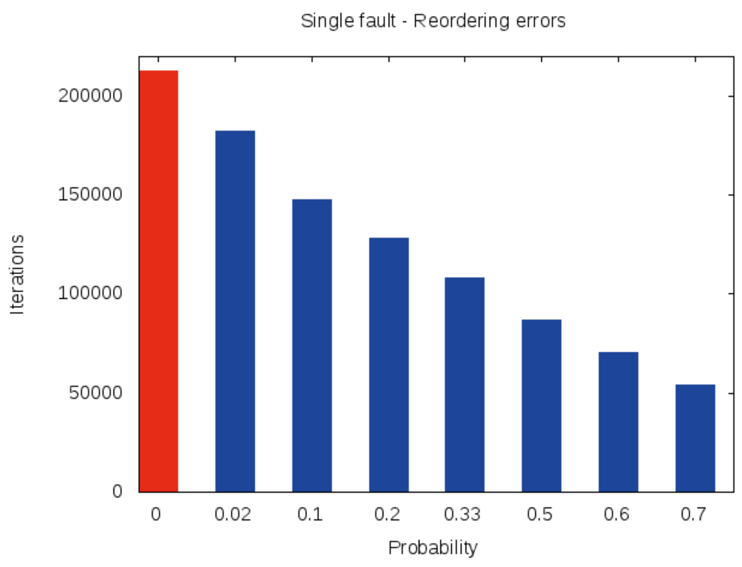


Figure 4.3: The figure shows the results of the third experiment. We test the first attempt algorithm for all error probability values, when only reordering errors occur. We note the lowest performance of the first attempt algorithm so far. This is an expected result of the tolerance for reordering errors of this first attempt version, as described in the original paper[4]. This time the performance goes below 50 percent already at the third highest probability value, 0.5.

4.2 Mixed error cases

For the mixed error cases, we subject both the first attempt and the full algorithm to each experiment. The probability value still indicates the chance of an error occurring, only this time it stands for the chance of occurrence for all possible errors. This means that each time an error is inserted we choose randomly between the possible errors in each experiment.

4.2.1 Omission & duplication

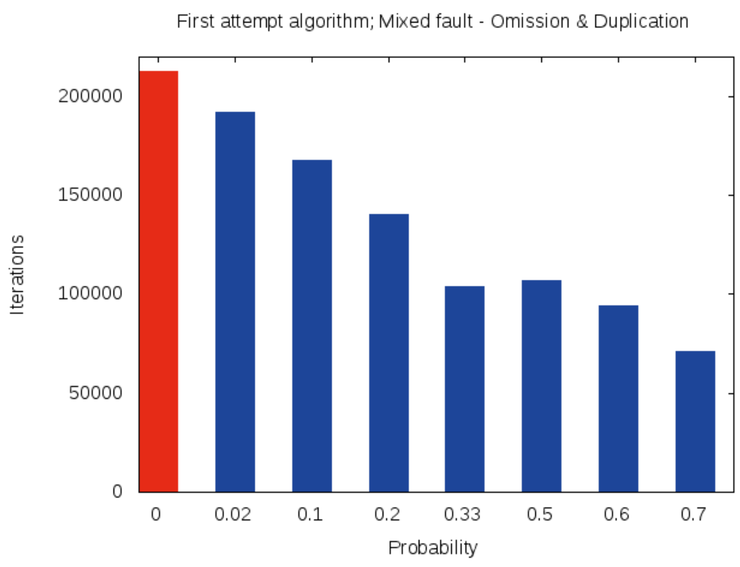


Figure 4.4: The figure shows the results of the fourth experiment. We test the first attempt algorithm for all error probability values, when omission and duplication errors occur. As expected, we see the characteristics of each error come back in this result, compared to the single fault errors with this algorithm version. The large overhead of the algorithm cripples it quite a bit when we look at this mixed error scenario.

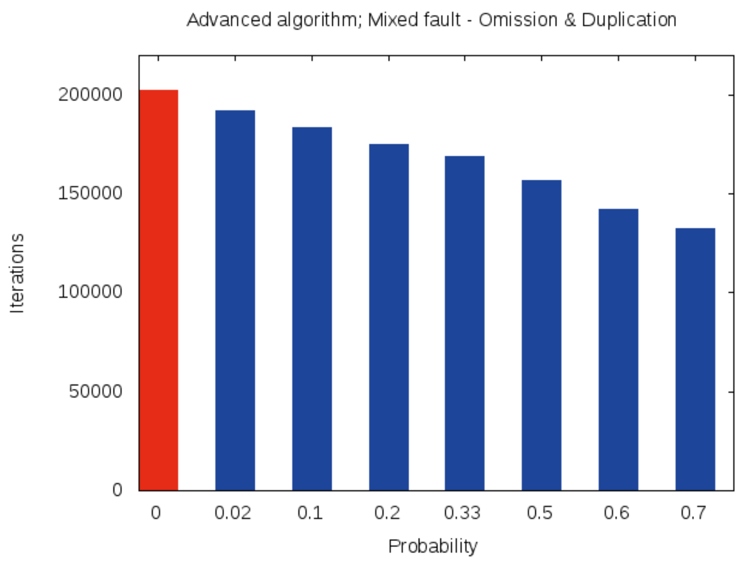


Figure 4.5: The figure shows the results of the fifth experiment. We test the advanced algorithm for all error probability values, when omission and duplication errors occur. Immediately we notice a very large improvement to the performance, compared to the first attempt result in the previous figure. The Error Correcting Code redundancy helps the algorithm when omission or duplication errors occur, by correcting them. The performance drops a bit toward the higher probability values, but it stays within 65 percent of the benchmark.

4.2.2 Omission & reordering

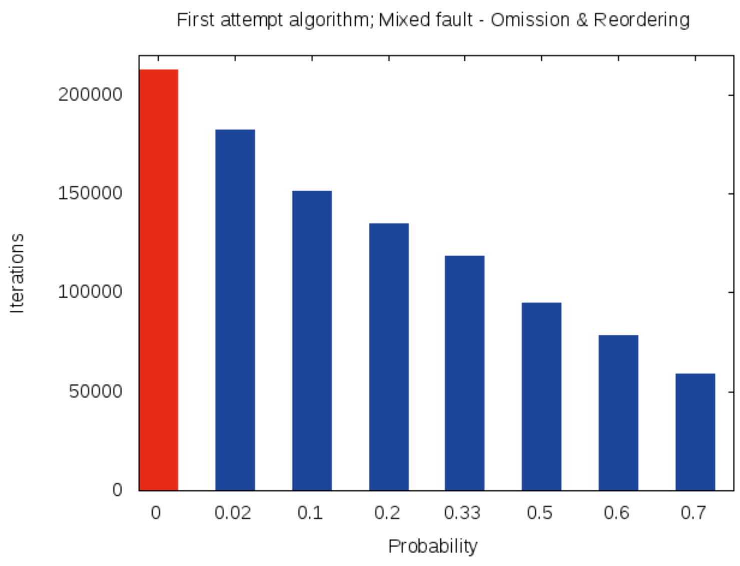


Figure 4.6: The figure shows the results of the sixth experiment. We test the first attempt algorithm for all error probability values, when omission and reordering errors occur. Comparable to the first mixed scenario, the performance of the first attempt algorithm goes down quite steep because of the high overhead. The omission errors are relatively easy to deal with, but the reordering errors really pulls the performance down.

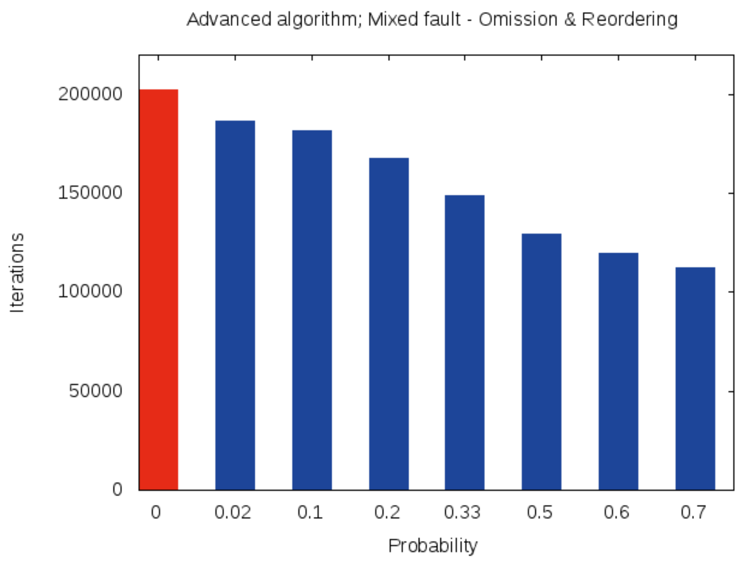


Figure 4.7: The figure shows the results of the seventh experiment. We test the advanced algorithm for all error probability values, when omission and reordering errors occur. We notice a lesser performance than the previous advanced algorithm experiment, especially towards the higher error probability values. Compared to the first attempt algorithm, the advanced version still outperforms it substantially.

4.2.3 Duplication & reordering

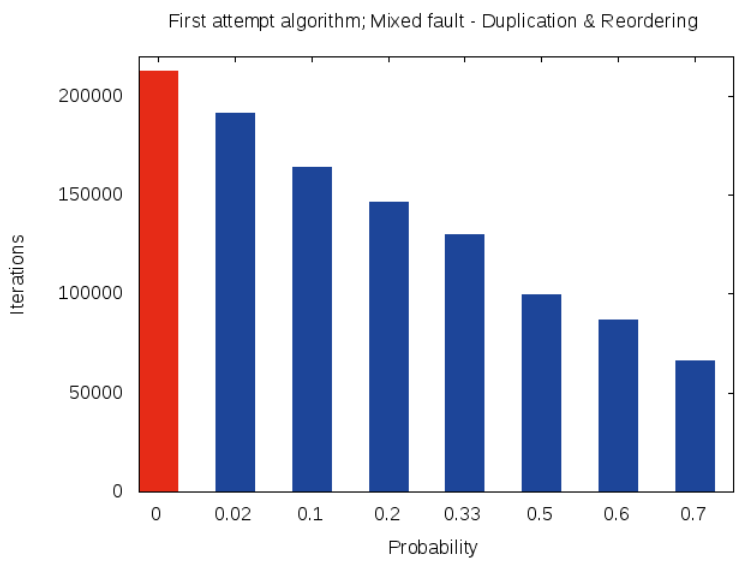


Figure 4.8: The figure shows the results of the eight experiment. We test the first attempt algorithm for all error probability values, when duplication and reordering errors occur. The low performance trend of the first mixed scenario experiments, continues again in this situation. The first attempt algorithm struggles with the high overhead, especially towards the higher error probability values.

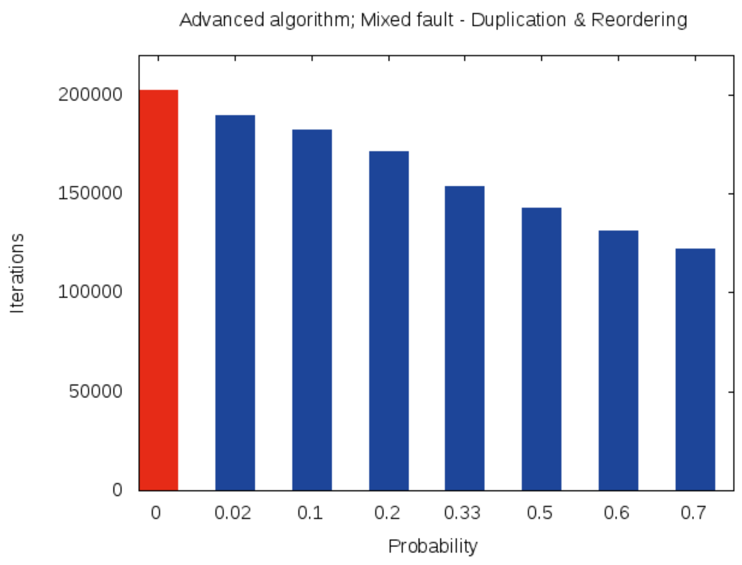


Figure 4.9: The figure shows the results of the ninth experiment. We test the advanced algorithm for all error probability values, when duplication and reordering errors occur. In comparison to the first attempt algorithm, the advanced version can yet again take full advantage of its error correction capabilities. In comparison to the other mixed fault scenarios, the performance is similar. Even at the highest error probability values, the algorithm's performance stays well above the 50 percent mark.

4.2.4 Combination of all errors

For this last set of experiments, we subject both versions of the algorithm to a combination of all possible errors. The probabilities stay the same, meaning the total amount of errors is a relative frequency of the amount of iterations. Each time an error is inserted, we choose randomly between the three possible errors.

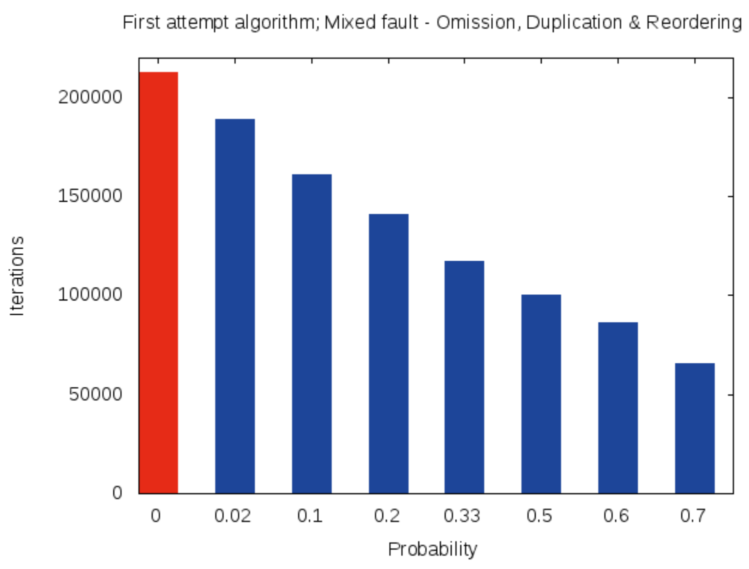


Figure 4.10: The figure shows the results of the tenth experiment. We test the first attempt algorithm for all error probability values, when omission, duplication, and reordering errors occur. This last result captures the essence of the first attempt version. It is capable to handle the errors but pays a heavy price for it in terms of performance.

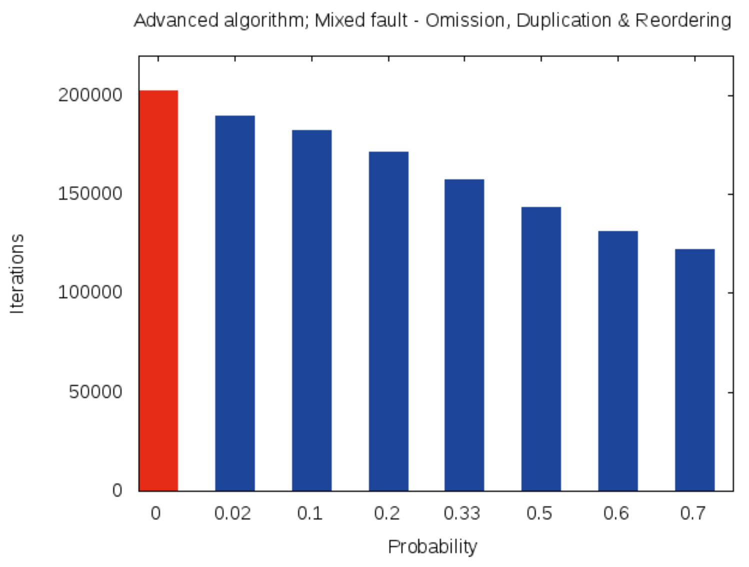


Figure 4.11: The figure shows the results of the eleventh experiment. We test the advanced algorithm for all error probability values, when omission, duplication, and reordering errors occur The performance stays above 50 percent at the higher error probability values, and even well above 75 percent for the majority of the simulations, at lower error probability values.

4.3 Analysis of results

After getting the data together and looking at the graphs, it is clear to see the advanced version of the algorithm works very well under pressure from errors. This is true even at high error probability values. The first attempt algorithm works, but that is all that can be said. The performance is very weak in comparison to the advanced version, because of the high overhead and low efficiency.

5 Conclusion

At the start of this project the research question was clearly defined. The theoretical advances had been made, now it only needed the practical proof of concept. We implemented a working version of the Self-Stabilizing End-to-End Communication in Bounded Capacity, Omitting, Duplicating and non-FIFO Dynamic Networks algorithm in TinyOS, according to the demands of the original paper in which it described the necessary steps to be taken for the practical version to be completely working.

We completed the practical demands that were needed in order to fulfill the self-stabilization criteria. We created two versions of the algorithm, ran the experiments, and assessed the performance according to an error probability constraint. Eventually we met the demands set forth by the self-stabilization criteria that specify the algorithm must withstand following communication errors concerning packet errors.

1. Omitting
2. Duplication
3. Reordering (non-FIFO)

The issues at hand concerning the development of such an algorithm in TinyOS are still relevant though. The debugging difficulties concerning sensor networks and distributed systems make it difficult at times to find a problem in the code.

In light of the results we established, and some of the problems we encountered, I look forward and take a moment to advise the future researcher on the possibilities that are still present in this work.

5.1 Future work

As mentioned in the beginning of this work; the algorithm was written as part of the research group Gulliver, who are doing research and activities concerning these types of sensor networks. The eventual goal for this algorithm is to be used in a working environment.

Second, the Reed-Solomon Error Correcting Code that I took from a library on-line could possibly be improved. After going through the research, Reed-Solomon was en still is the perfect candidate for this algorithm as an Error Correcting Code. The only problem is the high complexity; it would take a separate project to develop a Reed-Solomon ECC algorithm. Taking it from a on-line library, because of time constraints, meant I couldn't dive in the code and see if I could remove some parts or optimize some elements that could maybe improve the performance even more[7].

Finally, looking back at the many problems I faced during development concerning

TinyOS, I would personally advise future researchers to switch to more supported operating systems like Contiki-OS. It is my belief this would result in a faster development process. TinyOS has its place, but it has been catched up. I base my statements on my own experience, and on the statements made by the founder and main developer of TinyOS, Professor Philip Levis of Stanford University. A few years ago he wrote an extended article about his experiences during ten years of TinyOS development. Looking back, and looking at the current status of TinyOS, even he sees the logical trend of research moving their aim away from TinyOS because of the non-active development[12].

Bibliography

[1] Y. Afek and G. M. Brown. Self-stabilization of the alternating-bit protocol. In *Reliable Distributed Systems, 1989., Proceedings of the Eighth Symposium on*, pages 80–83. IEEE, 1989.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications magazine, IEEE*, 40(8):102–114, 2002.

[3] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[4] S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Stabilization, Safety, and Security of Distributed Systems*, pages 133–147. Springer, 2012.

[5] R. Fantacci. Generalised stop-and-wait arq scheme with soft error detection. *Electronics Letters*, 22(17):882–883, 1986.

[6] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[7] R.-S. C. D. Hobbs. Reed-solomon codes, 2011. [Online; accessed 12-December-2015].

[8] J. Hui, P. Levis, and D. Moss. Tep 125: Tinyos 802.15. 4 frames.

[9] P. K. Janert. *Gnuplot in action.* Manning, 2010.

[10] H. Karl and A. Willig. *Protocols and architectures for wireless sensor networks.* John Wiley & Sons, 2007.

[11] O. Landsiedel, E. M. Schiller, and S. Tomaselli. Libreplay: Deterministic replay for bug hunting in sensor networks. In *Wireless Sensor Networks*, pages 258–265. Springer, 2015.

[12] P. Levis. Experiences from a decade of tinyos development. In *OSDI*, pages 207–220, 2012.

[13] M. Pahlavan, M. Papatriantafilou, and E. M. Schiller. Gulliver: A test-bed for developing, demonstrating and prototyping vehicular systems. In *Proceedings of the 9th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '11, pages 1–8, New York, NY, USA, 2011. ACM.

[14] Y. J. Qazi, S. Muhammad, and J. A. Malik. Performance evaluation of error correcting techniques for ofdm systems. 2014.

[15] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. Deploying a wireless sensor network on an active volcano. *Internet Computing, IEEE*, 10(2):18–25, March 2006.

[16] S. B. Wicker and V. K. Bhargava. *Reed-Solomon codes and their applications.* John Wiley & Sons, 1999.

[17] T. D. Wiki. The tinyos printf library, 2010.