5 METHODS

Describing How You Solved the Problem or Answered the Question

5.1. Tools

As I discussed earlier, the work of my thesis has several goals. One of them is providing a solid starting point for a continuation of this research into future work, in case a replication of my experiments is needed, or in the case that the algorithm needs to be implemented in a bigger project.

In this part I'll discuss the tools I used to implement and evaluate the S^2E^2C algorithm. By giving a description of the used materials, future research can start from my exact situation. This way I provide all the information of this project in the form of this thesis. 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.

5.1.1 TinyOS

TinyOS is a free, open-source, flexible and application-specific embedded operating system designed for low-power wireless devices, specifically targeting Wireless Sensor Networks (WSN). It was developed by the University of Berkeley California and has since grown to be the TinyOS Alliance. The initial release was in 2000. At the time of writing the latest release, and the version I worked with to implement the algorithm, is version 2.1.3 and was released in 2012.

Wireless sensor networks usually consist of a large number of small nodes, which are designed to operate with strict resource constraints concerning memory, and power consumption. These limited resources and low-power characteristics require a smart operating system that can withstand these challenges and successfully handle high load operations while maintaining a certain level of quality in execution. TinyOS meets these challenges.

Often the comparison is made between TinyOS and other methods for developing applications on embedded systems. An example of one such method are the popular Arduino microcontroller boards. Arduino is more lightweight than TinyOS. The reason for this is simple; Arduino only offers some small scale, easy to build and modular kits, based on C code, for microcontrollers and sensors. TinyOS on the other hand, is a fully developed operating system. Consequently, while it is very easy to get started with Arduino, the learning curve for TinyOS is quite steep. For advanced and powerful applications, and large WSNs however, this pays off in TinyOS' advantage. Especially applications with elements such as multi-hop routing, reliable dissemination and time synchronization require an operating system that can efficiently handle these complex operations. tinyosfaq tinyoswiki praveen

5.1.1.1 nesC

TinyOS itself and its applications are written in nesC, a programming language specifically designed for networked embedded systems. It uses a event-driven programming model, and improves reliability while reducing resource consumption. It achieves this by having restrictions on the programming model. In their initial presentation of the language, the original developers acclaim that

> *"based on our experience and evaluation of the language it shows that it is effective at supporting the complex, concurrent programming style demanded by the new class of deeply networked systems such as Wireless Sensor Networks."* nesc

The use of this new programming language makes programming in TinyOS challenging. On one hand, nesC is quite similar to C. This means that implementing new features or protocols usually isn't hard. The difficulties arise when trying to incorporate new code into existing one. The part where nesc and C differ greatly is in the linking model. Writing software components isn't very difficult, but combining a set of components into a working application is quite challenging and complex. Unfortunately this is something that I was forced to do throughout the implementation process of my thesis. As I will discuss later in this report, this made my work quite difficult at times. tinyosprogramming

5.1.1.2 Hardware support

TinyOS can function on a number of supported hardware platforms. For my thesis, the hardware that was used is the TelosB Mote Platform. This is the platform used for most of the WSN research at Chalmers.

All of my algorithm tests and experiments were performed in the form of simulations. This means I did not use actual hardware. Instead, the simulator provided an simulation image of the TelosB mote. By doing this I was able to replicate real life experiments very accurately, since the compiler makes sure my application was compiled specifically for TelosB motes, by providing the `telosb` command as an argument to the `make`compile function. telosbspecs

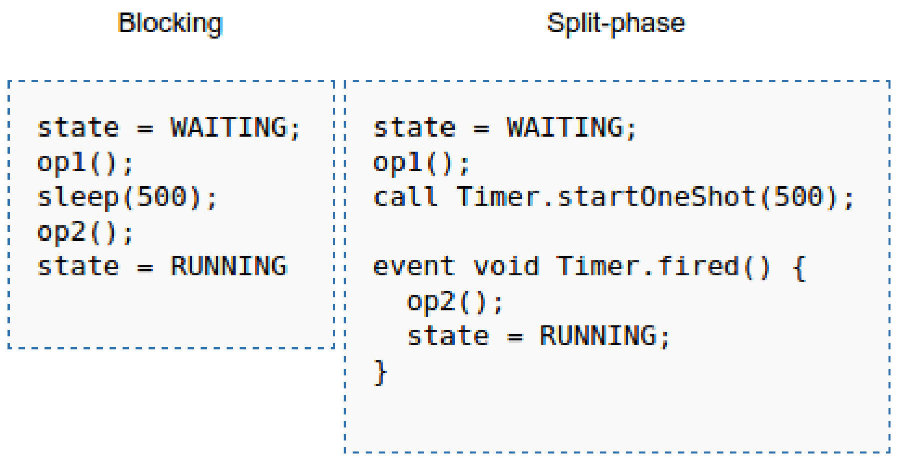
5.1.1.3 Execution model: split-phase

TinyOS has no blocking operations, because every long-running operation is executed split-phase. This offers significant advantages over a blocking system.

- Split-phase calls do not occupy too much stack memory while they are executing.

- The system stays responsive at anytime. All of the application threads are kept free from being held down by blocking calls.
- Split-phase operations reduce stack utilization.

A blocking system on the other hand, uses a different approach. Here, the you are never sure whether or not a call for a long running operation will return or not. The program will most definitely block because it has to wait until the operation is complete and the call returns. Split-phase systems on the other hand, return the call immediately. A simple example of the difference between the two is shown in the figure below.



```
state = WAITING;
op1();
sleep(500);
op2();
state = RUNNING
```

```
state = WAITING;
op1();
call Timer.startOneShot(500);

event void Timer.fired() {
  op2();
  state = RUNNING;
}
```

Figure 1: A split-phase operation example. While split-phase operating systems require more code to reach the same goal, they are far more robust.

splitphasesource

5.1.2 TOSSIM

TOSSIM is the standard simulator for TinyOS. The newest version can be found in the TinyOS source directory when you install it, since TOSSIM is a standard part of TinyOS. It simulates entire TinyOS applications, and works by replacing components with simulation implementations.

For my research I have not used TOSSIM. Instead I was instructed, during the explanation of my assignment, that I would use Cooja. This means I will not go into further detail on TOSSIM, but further details can be found on the wiki page tossimwiki.

5.1.3 Cooja

Cooja is the WSN simulator I used for my research. It comes standard with a version of Contiki-OS, and has a few advantages over TOSSIM. It has advanced capabilities, active support, and it can work with or without a GUI. This gave me more flexibility for simulating the algorithm. An added advantage is that the GUI allows for a smaller learning curve, which meant quicker testing and development.

Contiki-OS or in short Contiki, is the self pronounced “Open Source OS for the Internet of Things”. In contrast to TinyOS, it is being commercially developed and supported. contikihomepage

When it comes down to it, Contiki and TinyOS and different operating systems with the same goal: to develop applications with a small footprint, targeting low-power devices as used in WSNs. My application is developed in TinyOS for TelosB motes, but I’m using the Cooja simulator from Contiki. This means using a slightly different approach of simulating applications, but luckily it is not difficult at all. The following steps need to be taken to simulate code based on TelosB motes in Cooja.

1. Cooja is included in the Contiki CVS Git. Make sure you clone the most recent version of Contiki from Git.

   ```
   git clone https://github.com/contiki-os/contiki.git
   /destination-directory/
   ```

2. Compile the TinyOS application, by executing the appropriate make command in the project directory. For TelosB motes this means, executing the following command.

   ```
   cd /path-to-project-directory/
   make telosb
   ```

3. Go to the Contiki installation directory. Once there, change the directory to `tools/cooja/` (these steps can be done in one command) and run `ant`. This will start compiling Cooja. Once it is done, it will greet you with an empty simulation environment.

   ```
   cd /path-to-contiki-directory/tools/cooja/
   ant run
   ```

4. Select *File* → *New Simulation.* Give the simulation a appropriate name and select if you’d like to use a random seed on startup. Once everything is filled in, hit “Create”.

5. Select *Motes* → *Add motes* → *Create a new mote type* → *Sky Mote.*
6. In the “Contiki process/firmware” field, browse to the location where you’ve compiled your code using the `make telosb` command. If you have followed these steps so far, the location will be in `path-to-project-directory/build/main.exe`.

When following these steps, you will be able to simulate your code on the nodes in Cooja. Since Cooja is a part of Contiki, not all features of Cooja will work in combination with the TelosB code. During my research I did not encounter a problem because of this issue.

5.1.4 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 several reasons.

- 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.

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

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 inserting, duplicating and reordering packets on the fly.

5.1.5 Github

The most used Version Control System (VCS) for distributed revision control and source code management (SCM) on the internet is Github. It is based on Git, which is

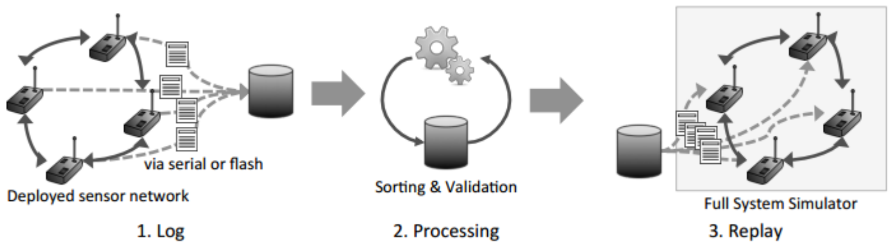


Figure 2: The above figure shows the way LibReplay works in two (three) steps: logging, processing the data, and replaying the exact state by feeding the algorithm all execution variables and necessary information to recreate the bug.

> *"a widely-used, free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency."* githubwiki

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. git

5.1.6 Atom

Atom was my preferred code editor for this implementation and subsequent research. This section may seem irrelevant, but I am mentioning it here because it has been vital for me in writing the code for the algorithm. The reason for this is that I searched very long for some kind of editor/IDE that had support for the nesC language. There is to the best of my knowledge unfortunately no other tool available that provides this coding support. At the time of writing, it is the only solution that supports the nesC programming language natively without requiring any other plugin or extra package. It is also a out-of-the-box working product for nesC, without requiring extra plugins or packages. atomeditor

5.2 Implementation of the algorithm

I divided the development of the algorithm in four phases. At the end of each phase I delivered a new version of the algorithm. I started the code from scratch, each new version building upon the previous one and adding functionality. The end goal was to test each version in a simulation environment, and subsequently compare the performance of each version compared to the other ones. I worked according to this structure, so that it was easy to quickly set up simulations and experiments for each version and compare them.

The name of each version is in accordance with the conventions as described in the original paper for my assignment ogpaper.

- First Attempt
- Advanced algorithm
- Full algorithm: advanced algorithm with Error Correction Encoding
- Full algorithm with multi-hop routing

All versions are based on point-to-point communication. This means a Sender sends his messages to one specific Receiver, and the Receiver acknowledges the messages back to the original Sender.

In the following sections I will highlight and explain parts of the code for each version. These are parts where either there was no immediate documentation for available, the provided documentation dated back to an older version of TinyOS, or I experienced many difficulties trying to implement it because of incompatibility or collision with other components.

5.2.1 First attempt algorithm

This is the first attempt version of the algorithm as described in the original paperogpaper. 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 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.

5.2.1.1 Printf debugging

As I described earlier in the part on LibReplay, the best and 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 tinyprintf.

The difficult part however, is reading the info 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
```

5.2.1.2 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;
```

5.2.1.3 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. compsandmods

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.

5.2.1.4 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.

5.2.2 Unit testing

5.2.3 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 divides the messages in packets and sends them to the Receiver. There is no implementation of Error Correcting Codes in this version yet. The point of this is to observe the performance of the algorithm without the Error Correction Codes, and then compare it with the performance of the algorithm once the Error Correcting Codes are added. This way it is much easier to see if the possibly increased performance of the algorithm thanks to the Error Correcting Codes weigh up against the added overhead that they bring along.

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.

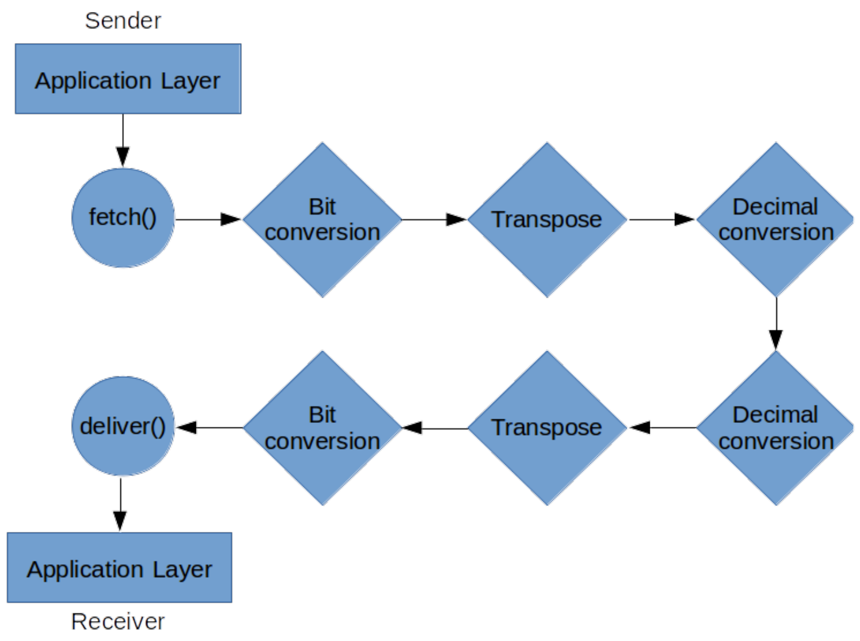


Figure 3: This figure shows the packet_set() function which is used to generate packets. It is clear to see that the sender and receiver functions are each other's opposites.

5.2.4 Error correction codes

This version contains the full sender and receiver algorithm as described in the paper. This is the first attempt algorithm, together with the packet generation functionality, and 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. eccvgl

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 drhobbs.

5.2.5 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 (Berkely 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. tep125

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);
```

5.3 Evaluation and preliminaries

1. The **performance** is measured through the time it takes to send a packet on its way to a receiver. This performance likely degrades as the algorithm becomes more complex. The reasoning behind this is that, with every added complexity (and new version of the algorithm), the overhead or expense to generate and send the packet, becomes larger. With every new version, the data gets edited more and subsequently it takes longer for the node to process the data and send the packet.
2. I measure as well the **amount of errors** that build up in the output of the Receiver node. An error can indicate either lost or corrupted packets and while these errors vary in terms of eventual impact on the application layer performance, the quality of transmission degrades nonetheless.

There are two variables in the algorithm that influence these criteria. The variable as indicated in the original paper, and the amount of nodes in the network.

1. The variable `CAPACITY` determines the amount of packets the Sender transmits to the Receiver, before fetching another batch of messages from the application layer. The prediction is that by increasing this number, the amount of errors will likely rise, because it will take more processing to Send the messages, and
2. The variable `SENDNODES` keeps track of the amount of Sender nodes that are used in the network. I've created this variable to make the algorithm easily scale according to the amount of nodes being used in the network. Since each Sender sends to one specific Receiver and vice-versa, the amount of Sender-nodes in the network (which should be equal to the amount of Receiver nodes) determines the address node id of the Receiver and Sender mote in each respective algorithm. My prediction is that the performance will again decrease the more nodes are used in the network. It is my belief that this will especially hold true for the implementation with the multi-hop routing algorithm, because of the complexity of the algorithm and the extra processing that's needed to get the packets to their destination.

5.3 Simulations

We want to achieve two things with these simulations. First of all, we'll test the performance of each algorithm in normal circumstances by letting the Sender transmit a very large amount of data to the Receiver node, multiple times in a row. Secondly, we'll run a sequence of simulations where we independently adjust the two variables we talked about earlier. First, we'll set `CAPACITY` to a fixed number. Then we will run a number of simulations in a row, and each cycle we increase the amount of nodes in the network, which will also change the variable `SENDNODES`. During each simulation, we measure the above mentioned criteria i.e., the performance and error count.

We'll repeat this procedure for values of `CAPACITY` from 1 to 16. The goal is to eventually find optimal values for both variables in which the performance and the error count balance each other out. We want to find the values that allow us the best performance while keeping the error count as low as possible.

5.3.1 Single-hop setup

To test the first three out of four versions of the algorithm we only need a single-hop setup between a sender and a receiver. The amount of total nodes in the network doesn't matter, as long as sender and receiver are in radio transmission range from each other.

5.3.2 Multi-hop setup

The multi-hop setup requires two extra nodes on top of the normal sender and receiver nodes.

1. **Root node**: in any RPL network, one node needs to be the root node. This node 'assists' the router in helping the packets arrive at their destination.
2. **PPP-Router**: this is a standard application in the TinyOS app directory. It provides routing control for multi-hop, IPv6 networks.

[Now we have routing; node 1 works as the root node and the PPP-Router performs it's routing functions to transfer packets through the network.][coojamultihop] [coojamultihop]: /home/evert/Documents/Thesis/Resources/IMAGES/coojamultihop.png
"Multi-hop operation example"

5.3.3 Automating simulations using test scripts

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.

I wrote one starting script, *simauto.sh*, and this is comparable to the key of a car; it starts the car but doesn't do much itself.

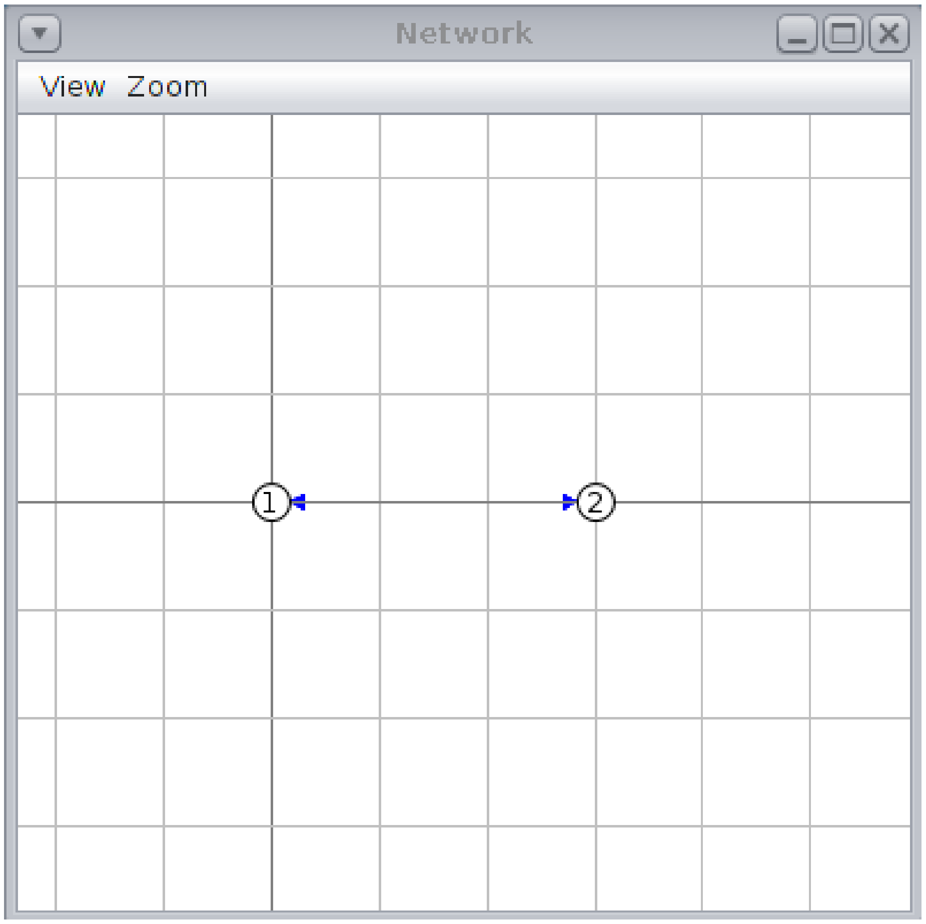


Figure 4: The above figure shows the single-hop setup between two nodes in a Cooja simulation. The blue arrows indicate bi-directional traffic.

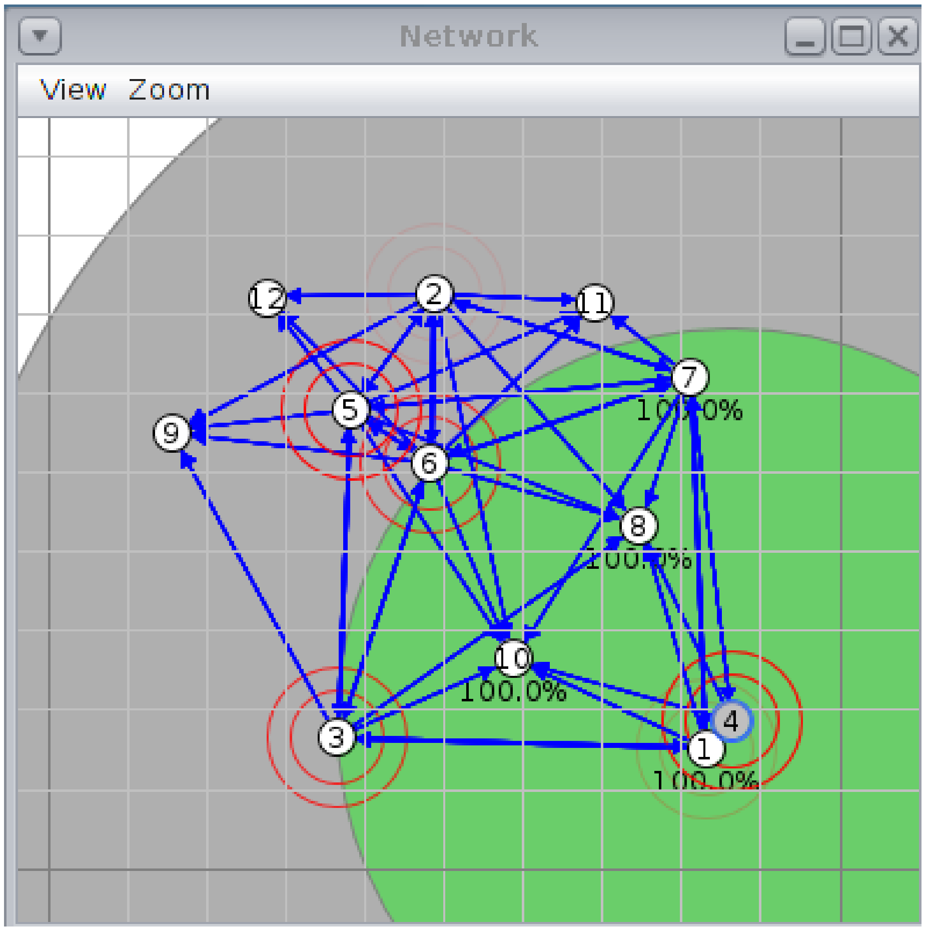


Figure 5: We add more nodes to the single-hop network. Now the bi-directional communication is clearly visible. Also notice that, because there is no routing yet, the sender and receiver nodes send their messages in all directions.

```
. /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 (this means Cooja works through the command line, without a graphical user interface) 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 has to do again with the non-sequential way that sensor networks work. These scripts need to be called at the same time, because you can't execute the algorithm and sequentially listen to the serial port; you won't receive any data.

5.4 Experiments

The experiments using LibReplay have one goal: to provide indefinite proof of the self-stabilization criterion that is discussed in the original paper. The criterion for self-stabilization requires that, starting from any arbitrary state, the system can return to a stable state in which communication goes smooth again. Practically this results in the following experiments that need to be done.

1. Insert,
2. duplicate,
3. reorder packets during algorithm execution.

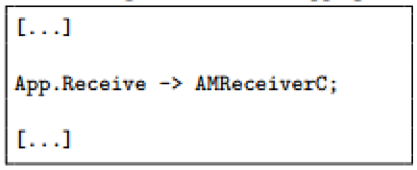
The goal is to show that, despite disturbing the communication by changing the packet transmission, the receiver will be able to correctly and swiftly deliver the data to the application layer.

5.4.1 LibReplay setup

LibReplay works quite simple in itself. Unfortunately TinyOS doesn't always work without problems with the features provided by LibReplay. As discussed earlier, LibReplay works in two steps;

1. Implement logging components on the interfaces you wish to log. Subsequently, LibReplay logs the function calls and current environment variables. The algorithm execution state is completely logged and kept track off.

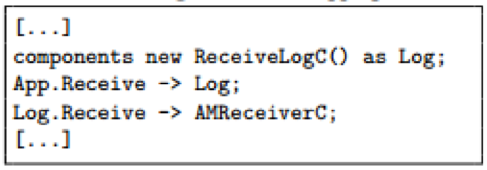
2. Using the same interfaces we logged in the first step, implement the replay components. This way the exact execution state can be fed back to TinyOS and we can replicate the exact circumstances in which a certain bug presented itself. Implementing the replay components is just a matter of replacing the logging components with the replay components.



```
[...]

App.Receive -> AMReceiverC;

[...]
```



```
[...]
components new ReceiveLogC() as Log;
App.Receive -> Log;
Log.Receive -> AMReceiverC;
[...]
```

Figure 6: The above figure shows the replay component added to the receiving interface. For our algorithm we'll log both the sender and receiver interfaces at both the sender and receiver motes.