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. 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 variables of this project in the form of this thesis.

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 an international consortium, 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. Examples include, but are not limited to, sensor networks, ubiquitous computing, personal area networks, smart buildings and smart meters. 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 and quickly became the platform of choice for sensor network research.

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. At at higher level, Arduino is lighter weight than TinyOS. The reason for this is simple; Arduino only offers some simple C support 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 a decent operating system that can efficiently handle these complex operations. tinyosfaq tinyoswiki praveen

nesC
TinyOS itself and it's applications are written in nesC, a programming language specifically designed for networked embedded systems. It uses a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design. nesC improves reliability and reduces resource consumption by including data-race detection and aggressive function inlining. It achieves this by having restrictions on the programming model. In their initial presentation of the language, the original developers acclaim that based on their 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 codes into existing ones. 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

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. It is a low power Wireless Sensor Module, and it is composed of the MSP430 microcontroller and the CC2420 radio chip. The microcontroller of this mote operates at 4.15 MHz and has a 10 kBytes internal RAM and a 48 kBytes program Flash memory.

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 compile function. telosbspecs

5.1.2 TOSSIM

TOSSIM is the standard simulator for TinyOS. The newest version can always 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. Hence, make sure you have a recent cvs checkout git clone of Contiki on your system.

   ```
   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

5.1.5 Scripts

All simulation and test scripts were developed in bash, the standard Linux scripting environment.

5.1.6 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 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. 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 githubwiki

5.1.7 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. At the time of writing, there is to the best of my knowledge unfortunately no other tool available that provides this coding support. It is the only out-of-the-box solution that supports the nesC programming language natively without requiring any other plugin or extra package. atomeditor

5.2 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.

- First-Attempt
- Advanced-No-ECC
- Advanced-ECC
- Advanced-ECC-Multihop

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.

5.2.1 First-Attempt

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

5.2.1 Advanced-No-ECC

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.

5.2.1 Advanced-ECC

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.

5.2.1 Advanced-ECC-Multihop

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'). As you will be able to see in the code, this brings along significant changes 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).

5.2.1 First attempt algorithm

5.2.2 Unit testing

5.2.3 Packet formation from messages

5.2.4 Error correction codes

5.2.5 Multi-hop routing algorithm

5.3 Simulations

5.3.1 Single-hop setup

5.3.2 Multi-hop setup

5.3.3 Automating simulations using test scripts

5.4 Experiments

5.4.1 LibReplay setup