Performance of WSN MAC Protocols in Congested Radio Environments

Manuel Weber

Manuel Weber B.Sc.

Performance of WSN MAC Protocols in Congested Radio Environments

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dr. Carlo Alberto Boano
Institute for Technical Informatics (ITI)

Graz, June 2016

Manuel Weber B.Sc.

Performance of WSN MAC Protocols in Congested Radio Environments

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

Dr. Carlo Alberto Boano
Institut für technische Informatik (ITI)

Graz, im Juni 2016

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Date/Datum	Signature/Unterschrift

Abstract

The Internet of Things is becoming an integral part of our daily lives, spreading into many different domains such as smart cities, health care and disaster prevention. Wireless Sensor Networks are used as enablers for the Internet of Things. For safety critical applications it is crucial that every packet arrives, despite interference. Nowadays interference is omnipresent, as WiFi and other wireless technologies populate the same frequency band used for communication in Wireless Sensor Networks. Therefore it is necessary to understand the influence of WiFi interference on communication, which is acquired by experimentation.

As platform the Tmote Sky equipped with the popular CC2420, running Contiki is chosen. ContikiMAC is examined through experiments with interference generated using JamLab. The focus of the evaluation is reliability. Due to insufficient data gathering tools provided with Contiki it is necessary to develop a framework which aids researchers in running experiments. Using the new framework, weaknesses of ContikiMAC are exposed and subsequently corrected. Following that the performance of ContikiMAC is increased under interference using Forward Error Correction and Jamming Agreements.

Kurzfassung

Das Internet der Dinge wird ein immer größerer Bestandteil unseres Lebens und übernimmt Anwendungen in den Bereichen von Smart Cities, Gesundheit und Katastrophenschutz. Drahtlose Sensornetzwerke werden verwendet um diese Anwendungen zu verwirklichen. Bei sicherheitskritischen Anwendungen ist es notwendig, dass jedes Packet, trotz Störungen ankommt. Da WiFi und andere Drahtlos-Technologien im Frequenzband leben, welches für die Kommunikation in drahtlosen Sensornetzwerken genutzt wird, ist es notwendig den Einfluss von WiFi Störungen auf die Kommunikation innerhalb solcher zu verstehen. Dies wird mittels Experimenten mit Störungen generiert durch JamLab erreicht.

Als Plattform für die Experimente werden die Tmote Sky mit dem beliebten CC2420, mit Contiki als Betriebssystem genutzt. Der Fokus der Experimente und deren Auswertung liegt auf der Verlässlichkeit der Kommunikation. Da Contikis derzeitige Tools zur Auswertung und Aufzeichnung von Daten unzureichend sind, wird ein Framework entwickelt, welches bei der Ausführung von Experimenten mit MAC Protokollen unterstützt. Mit Hilfe dessen werden Schwächen in ContikiMAC festgestellt und daraufhin behoben. Darüber hinaus wird die Performance von ContikiMAC durch den Einsatz von Forward Error Correction und Jamming Agreements verbessert.

Contents

Acknowledgements

I want to thank my supervisor Dr. Boano who always had time to help me even though he had so little. I also would like to thank Markus for his encouragement and being such a good friend for so many years. I would like to thank my girlfriend for motivating me and my friend Daniel for being there for me when I needed him. Other than that I want to express my gratitude towards my parents who supported me all the way to the very end. And there's coffee of course.

Manuel Weber
Graz, Austria, June 2016

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews, 2012].

Chapter 1

Introduction

The Internet of Things (IoT) is a rather new, vastly growing field. The vision of IoT is: as more and more of our daily objects we use gain access to wireless communications (smart phones, smart watches, sensors, Radio-Frequency IDentification (RFID) tags and many more), they will gain the means to communicate with each other, exchange data to work towards a common goal [Atzori, Iera, and Morabito, 2010]. These goals vary and infiltrate our very lives in many different ways.

As mentioned before the IoT is growing fast with more and more applications being created while new hardware is developed at lightning speed. Examples for IoT applications show a wide variety: from personal use, over home automation to smart cities. There are several different fields which experience being redesigned and made ready for the IoT, each of which has its own benefits and challenges:

- **Smart City**: Cities where tiny sensors installed, for example, in the ground, provide information, that can make daily life more comfortable as well as save energy in different fields [Zanella et al., 2014]:
  - Smart parking is one of those applications. Every driver in a city has experienced the frustration of looking for free a spot on a busy day. An app which tells the user if parking spots in an area are taken or not would be beneficial not only for the mood of the driver but also for the environment, as less time spent looking for a free spot equals to less pollution [Chinrungrueng, Sunantachaikul, and Triamlumlerd, 2007]. There are different solutions for this problem: sensors in the ground beneath a parking lot communicate if it is occupied to a base station.
  - On-demand garbage collection can save a lot of energy and money for the companies employed by the city and frustration for the residents. At the moment many cities have a scheduled garbage collection, which can be made more efficient by placing sensors in garbage bins which communicate if they have to emptied. The companies themselves can then find the optimal way to collect the garbage.
  - Air quality: An application which can tell users where the air is best in the city for outdoor activities as well as alarm the authorities if pollution levels in certain areas should be reduced, could be realized by installing small sensor devices (sensor nodes) which communicate the current air quality to a base station.
  - Smart Lighting: The current paradigm of lighting systems in cities uses continuous illumination of streets and bigger places [Müllner and Riener, 2011]. Most of the time the need for light during the night time is only short. Other than replacing current lamp posts with LED lighting, posts would also need sensor nodes which would act as actuators, as soon as a person is detected within its radius by for example, GPS via cellphone.

  - Water quality control: Sensors placed in the water monitor different quality indicating attributes, and communicate these values back via a wireless network [Le Dinh et al., 2007]. This can be used to not only detect hazards as fast as possible, but also to detect leakages faster, a process which often takes time as only a big part of a water network can be detected automatically to have a leakage. Automating the whole process might result in water savings, which would be helpful for many big cities.

  There are many more applications for smart cities, but the examples given above should give the reader a good idea about the direction cities and their technologies will move to.

- **Smart Home**: Home automation has come a far way. Temperature control, cleaning robots, shutters that react to light intensity or wind velocity are some examples. The next step is to be able to control all these different system over the net. Security is an important concern of users. Nobody wants a stranger to be able to control one's home. The system should use little additional energy or save energy altogether.

  Temperature control is something that has existed for quite a while now, with a central control and maybe a remote control. Now there would be several sensors, enough to have a good estimate of the room temperature which send the data to a central device which then decides based on for example daytime which temperature to achieve. The user can retrieve the data from central device via cellphone or computer and decide whether the standard values are good enough or not. The user also can decide to put other sensors like outside temperature, humidity and wind into the equation of the inside temperature.

  Most of the aforementioned devices will communicate over the air and will have to compete with WiFi and the microwave, both of which exist in the same frequency band. Therefore it is necessary to ensure little energy consumption together with reliable transmissions.

- **Smart Health**: Wearable devices that monitor the health status to your cellphone or in case of an emergency to the ambulance have existed for a while now. In this field miniturization led to many different possibilities:

  - Implanting devices under the skin which can monitor blood or release contraceptive [Review, 2014].
  - Training exercise gadgets like socks or watches which synchronize to a smart phone to show how well the training session went
  - Sensors implanted in the bed which monitor a person's sleep and sends the date to its smart phone [Beddit, 2015].
  - Heart monitors which automatically send emergency calls if an anomaly happens.

  For these devices it is often more important to reliably deliver the information than the delivery speed. For some of the devices, for instance the implanted contraceptive or blood sugar monitor, energy efficiency is essential. This means that unnecessary transmission should be minimized, meaning the sending device should always know if its data arrived at the destination or not.

- **Smart Grid**: As in other sectors, the 'smart' revolution did not stop here. At the moment most of the plans for the smart grid are still being implemented slowly. The goal is to have a grid which is self-aware, and can react to changes itself. Smart Meters at every end of the connection enable two-way communication; the user him/herself can choose which tariff to use, while reading the meters does not require an employee anymore. Sensors on every every transformer can tell a company how the state of its hardware is and inform quickly if there is a power outage.

  While it may seem like it would not be important for these devices to run with little energy, its purpose actually is to save power and work. This means that even though energy might not be

essential, saving energy still is an important goal. Moreover most of the communication will still be over the air, as installing wired connections would be impracticable. The need for proper communications protocols which are reliable yet energy-saving can be seen in this use-case as well.

- **Disaster Detection**: In recent years sensors were installed in endangered areas to monitor the potential for disasters like wildfire, landslides or volcanic activity. For example, Ramesh et al [Ramesh, 2009] build a wireless sensor network, a network of tiny battery-powered computers, to monitor soil of an area to estimate landslide risks. In their field tests they installed 20 sensors nodes with a lead battery which was recharged by a solar panel. The information gathered by the sensor nodes is then forwarded to a gateway via wireless communication. The gateway itself is a more powerful computer which is equipped with WiFi which is than used to transmit the data to a central server. The data of the sensor nodes is then analysed on the server.

  The schema described above is often used to monitor large, remote areas where a steady power supply is hard to achieve and in order to reliably monitor the whole area, many sensors are needed. In this case as well it is important to reliably deliver information while the latency is not that important.

The Internet of Things (IoT) has many different applications and many of those are based on Wireless Sensor Networks (WSN) which build the backbone of the IoT. It is important not to confuse these two terms. WSNs are a key component and key enablers of the IoT which strives to connect everything to internet. Many of the examples above consist of WSNs but not all of them. Some just need a connection to a remote control and had better not be connected to the Internet (for example, the implanted contraceptive). The focus of this thesis lies on WSN and its communications. To be more specific the focus will lie on the reliability and energy-efficiency of the communications within WSNs.

1.1 Problem Statement

The examples provided have a theme in common: small microprocessors with little power supply need to communicate their data, mostly using wireless communications, sometimes in harsh environments. These so-called sensor nodes should be long-lived and need almost no maintenance, as changing batteries is often as cost-intensive as redeployment, if we think of harsh environments. However, sometimes just saving energy is not enough: the data which is monitored and transmitted is important and potentially life-saving, so it needs to be transmitted reliably, to be ensured to arrive at the destination.

If we think back to the examples of WSN for disaster detection, it is desirable to ensure that every data communicated from the sensor nodes reaches the base station, as data loss could potentially cost lives. This will lead to retransmissions if a packet is lost due to external factors which will in turn cost more energy, but losing packets is not an option.

A quite different example would be smart parking applications in cities. Sensors are in the ground and communicate if nearby parking spots are occupied or not. Exchanging batteries might be very cost intensive and induce the need to close off the street while the procedure is ongoing. This in turn costs additional manpower which makes it even more expensive if batteries run out. While it is annoying if a sensor node fails to transmit the status of its parking lot it will not endanger lives, which means that it is desirable to have a high success rate when transmitting data, but it is more important to save energy, a trade-off which is not avoidable.

As final example smart health devices like heart monitors or blood sugar measurement devices come to mind: The delivery of every packet is essential, it could be an alarm which would save a person's life. No external factors, be it WiFi streaming of video or music, which happens more often in homes these days, or a micro-wave oven which was switched on just before the blood sugar was dropping, should be able to keep the packet from being delivered. It does not matter if the packet is delayed for half a second

in those cases, but it absolutely has to be delivered. At the same time energy should not be wasted, meaning therefore the sender should know if the packet was transmitted successfully, in order to prevent unnecessary retransmissions.

The focus of this thesis lies on being able to reliably transmit data despite interference, while using as little energy as possible. This means that whenever data has to be transmitted it should be delivered and acknowledged. The acknowledgements serve the purpose of giving the sender correct information of the state of the packet. This can be used to save energy as information does not have to be resend after an ACK is received. In general when no data has to be transmitted, very little energy should be used for the protocol. This will be the case most of the time for many safety critical applications. These send regular updates, with the occasional alarm which is sent if a threshold is passed.

These challenges can be solved on the Media Access Control layer where connection or link between the nodes is managed. There the radio is directly controlled using the driver and which additional data concerning the connection should be sent. Currently there are several different methods to mitigate interference and some of them might work well together while others will not. The aim of this thesis is to understand the effects of interference on MAC protocol performance on a detailed level. Through the gained knowledge counter measures should be found and applied, and the effects of those will then be evaluated.

1.2 Approach

In this thesis, the followed approach is experimental computer science. Contiki's default protocol, ContikiMAC, is being tested, under reproducable conditions in environments rich of radio interference. To ensure comparability among protocols, a repeatable interference generator is needed, for which JamLab was chosen. The different interference sources will consist of different types of influence, like video streaming via WiFi, Microwave ovens and file transmissions. These are typical interference patterns which can happen in a work or home environment.

ContikiMAC will be evaluated by the following metrics:

1. reliability of the communication and
2. the total energy consumption used for a given number of packets transmitted.

After evaluation weaknesses exposed through the experiments will be solved and the enhanced protocol will be re-evaluated. In order to achieve these goals and to enable fast development of the protocol, a test-framework has been developed which eases testing as much as possible.

The experiments are carried out in the testbed located at Technical University Graz, ITI which is equipped with Tmote Sky sensor nodes. As operating system for the sensor nodes Contiki (see Section 2.2) has been chosen, due to its popularity and widespread use.

1.3 Limitations

The focus of this thesis lies on MAC protocols, their energy consumption and reliability, which will be evaluated on a single link. Energy consumption and reliability are not the only important factors. For a video surveillance system latency and throughput might be important, while energy consumption and reliable communication might be of less interest.

Another way of combating interference is to adjust the Routing layer, which controls which nodes are used to send packets from one node to another. Adaptive routing could also be used to ensure higher reliability and lower energy consumption, but routing relies on the underlying MAC protocol to provide stable links.

The energy consumption of the nodes is not directly measured, but rather estimated by Contiki's inbuilt Energest, which provides a good estimate. It would be another work to develop an infrastructure which could directly and accurately measure the energy consumption of every node in a testbed, which is why Energest, which is cheap and easy to use, is the tool used for energy consumption measurement.

Although JamLab provides repeatable interference patterns, recorded from real interference, it still is not real life interference, which is a mixture of many different devices using the same frequency band. JamLab is periodic, real life interference is not. JamLab still is a good representative of the classes of interference which can be found in homes, offices and is therefore used as interference source.

1.4 Contributions

This thesis contributes an improved statistic system to Contiki which is ready to use with the CC2420 and ContikiMAC, as well as a framework which assists in running experiments and evaluating MAC protocols in Contiki for the Tmote Sky sensor node.

Additionally this thesis contributes a more resilient version of ContikiMAC to WiFi interference, enhanced with Forward Error Correction (FEC) and more reliable acknowledgements using jamming agreements.

1.5 Structure of this thesis

In Chapter 2 the necessary theoretical background is given in order to be able to understand the following chapters of this thesis. Chapter 3 introduces the test-setup and shows the results of the first experiments. The results led to the development of a testing framework, explained in detail in Chapter 4. Using this framework we will revisit our previous experiments in Chapter 5. The new insight leads to the development of a new protocol introduced in Chapter 6, its evaluation follows and the protocol is refined and adapted in Chapter 7. Finally related and future work is given in Chapter 8.

Chapter 2

Background

Wireless Sensor Networks (WSN) are an integral part of the IoT and their correct operation is of utmost importance in several application areas. The focus of this thesis lies on the operation of WSN MAC protocols in congested radio environments: as we will show in the next chapters, their performance can largely affect the reliability and energy-efficiency of the overall network.

In order to introduce the reader of this thesis to these topics, we first provide an overview of wireless sensor networks in Section 2.1, with specific emphasis to common hardware platforms (Section 2.1.1) and operating systems (Section 2.1.2) in use nowadays. We then specifically describe the Contiki operating system in Section 2.2, as we choose this as the main software platform for our experiments. We provide an overview of MAC protocols in Section 2.3 by discussing the different strategies that have been developed by the research community and describe the implementations that are available in Contiki in Section 2.4. We conclude this chapter with an introduction to Forward Error Correction (FEC) in Section 2.5. As we will show in the following chapters, this technique can significantly help in increasing the performance of MAC protocols in congested radio environments.

2.1 Wireless Sensor Networks

Wireless Sensor Networks (WSN) are networks consisting of small, power efficient micro chips with a radio module. They communicate on the ISM frequency band and are mostly battery powered. WSNs are often used to gather data which is then forwarded to a base station, which then processes the data.

2.1.1 Hardware Platforms

The hardware platforms used in WSN are typically power efficient micro-controllers with very constrained ROM and RAM size (see Figure 2.1). Additionally they are equipped with several different sensors, such as temperature, humidity or light sensors. Finally to be able to communicate the sensor

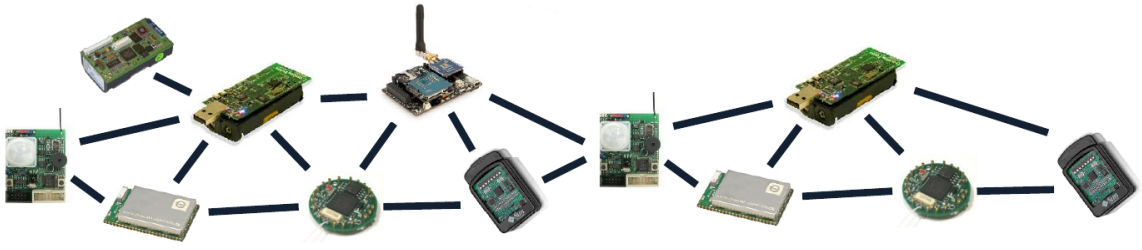

Figure 2.1: A collection of different WSN hardware platforms [Image gratefully extracted from Boano [2015a, page 2]]

node is equipped with a radio module, which commonly consumes most of the power when used. Sensor nodes are typically battery-powered and often the batteries are bigger than the sensor node itself. If sensor nodes are deployed in the wilderness or remote areas energy harvesting, like solar panels are used to ensure a long lifetime of the sensor nodes.

We now describe two examples of wireless sensor nodes: the first generation of "motes" (i.e., the TelosB or Tmote Sky nodes released in 2004) and one of the latest (i.e., the TI SensorTag CC2650 released in 2015). The Tmote Sky sensor node is the one used throughout this thesis and, despite its age, it is still a popular sensor node deployed in many testbeds.

2.1.1.1 Tmote Sky

The Sentilla Tmote Sky [Advantic Sys, 2015], [Moteiv Corporation, 2006] is perhaps the most widely-used wireless sensor node: the original design from Berkeley University and a number of replicas from different manufacturer have been on the market for quite a while. The Tmote Sky comes with:

- a TI MSP430F1611 with 48KB programmable flash, 10KB of RAM and a clock speed of 8MHz. Its current draw is 330μA in active mode and 1.1μA in standby mode [Texas Instruments, 2015b].
- a TI CC2420 2.4GHz RF transceiver module which is IEEE 802.15.4 compliant. Its current draw is 18.8mA in RX mode and 17.4mA in TX mode.
- two different light sensors (560nm and 960 nm peak sensitivity), as well as a temperature and humidity sensor.
- a 12-bit ADC, an UART via USB which also serves as programming interface.

The Tmote Sky nodes are supported by all major IoT operating systems (Tiny OS, Contiki, RIOT) and have been very popular in the last decade. The CC2420 radio has been used very widely, due to its compliance to IEEE 802.15.4 and its ability to duty cycle [Buettner, Gary V. Yee, et al., 2006] and even though the Tmote Sky has little RAM to use it is able to run an IPv6 stack with 6LoWPAN and Contiki as underlying OS [Durvy et al., 2008]. The Tmote Sky can be powered by either by USB or by two AA batteries. The battery lifetime depends mainly on the use of the radio module and its duty-cycling. At the moment the Tmote Sky itself isn't available anymore, but there are several replicas that have been produced. The version which is used for the experiments conducted in this thesis is the MTM-CM5000 v0.2 manufactured and sold by Advanticsys.

2.1.1.2 TI SensorTag CC2650

The TI SensorTag (see Figure 2.2) is a series of popular boards that have been attracting several developers in the past years. Its latest version, the CC2650 [Texas Instruments, 2015a], was chosen at the Institute for Technical Informatics of TU Graz for several courses, such as the Embedded Internet laboratory [Boano, 2015b].

This sensor node features:

- ARM Cortex-M3 with up to 48-MHz clock speed, 128KB in-system programmable flash, 28KB of SRAM.
- Ultra-low-power sensor control which can run autonomously from the rest of the system.
- 2.4-GHz RF transceiver compatible with BLE 4.1 and IEEE 802.15.4 physical and MAC layer.
- Peripherals like four general-purpose timer modules, 12-Bit ADC with 200k samples/s, UART, AES-128 module, and many more.

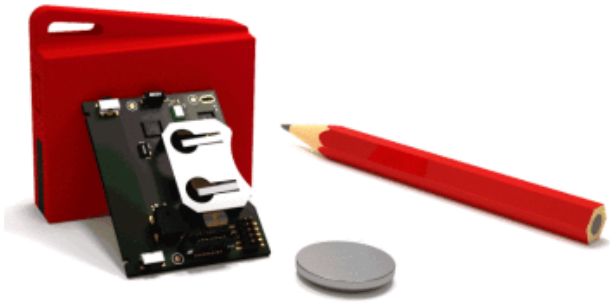

Figure 2.2: An image of the SensorTag CC2650 with sleeve [Image gratefully extracted from Texas Instruments [2015c]]

The SensorTag can further be easily extended with several development packs [Texas Instruments, 2015d]:

- a debug devpack that adds a JTAG debug interface
- a watch devpack that contains a small 96 x 96 ultra-low power graphical display
- a LED audio devpack that contains 4 high-power multicolor leds and a 4W audio amplifier

The SensorTag CC2650 was released in 2015 and can be acquired for € 29 and is hence significantly cheaper than the Tmote Sky. Despite being rather new, the CC2650 SensorTag is already supported by several IoT operating systems such as Contiki and constantly maintained. While preparing the Embedded Internet laboratory and this thesis, we found several bugs in the Contiki implementation [GitHub, 2015] that were filed and fixed.

2.1.2 Operating Systems

The WSN community grew rapidly and soon the first embedded operating systems, which have little to do with real-time operating systems, emerged. Developing bare metal is a nightmare as anyone who had to do that can attest. It takes very long to read into the technical specifications of a board and development for just one platform is very inefficient. Code should be reusable easily and driver- and platform-development should be well separated from application development. While that may introduce some inefficiencies, the gain in development speed and the decrease in code complexity is invaluable. Moreover most of the application developers do not want to touch low level code. They would rather concentrate on their applications or communication protocols.

So what is important for a WSN operating system? While it has to provide hardware abstraction like a general purpose OS (Linux, Windows), it also has to deal with heavily constrained devices like the Tmote Sky or the sensortag. Because of this, dynamic memory and threading are often not fully, if at all, provided by the OS. This introduces different approaches to the problem of (semi-) parallel execution and its programming.

We now describe a few OS that are popular in the WSN community. We start with Tiny OS, which is the oldest. Tiny OS's approach was to minimize the code size and optimise energy consumption and execution overhead. Therefore Tiny OS chose an event driven approach to its scheduling. Afterwards Contiki, a widely used IoT operating system, with its semi-threaded approach will be introduced. Contiki focused more on the ease of development for its platform than on raw efficiency. Finally, as an example of a very recent operating system, RIOT is introduced which features full multi-threading support.

2.1.2.1 Tiny OS

Tiny OS [Levis et al., 2004] is one of the first WSN operating systems. Its development started at USC Berkeley but was soon made open source and several other research groups (CMU, UCLA, UIUC, Vanderbilt) started to contribute.

Tiny OS is event based and highly modular. Concurrency is also event-driven. Tiny OS possesses a task-queue which contains all the tasks the scheduler has to execute. The tasks themselves are only function calls which can be pre-empted by interrupts which in turn can trigger hardware events. Those can add additional tasks to the queue. If all tasks in the queue have been executed, which leaves the queue empty, the sensor node goes to sleep and resumes execution on the next interrupt. Tasks can be encapsulated into a named component (or module) together with a state. Every component has interfaces which it uses to communicate to other components. Interfaces can either be commands or events a component can handle. Commands are requests, events signal the completion of such a request. A Developer can specify an application by taking several components and connecting their interfaces. This process of programming reminds of the way one would program with hardware description languages, such as Verilog.

While this approach is very efficient in execution and memory, it also gets very complex with the number of components used. This is caused by the necessity of an explicit program state which is interacts with several components. Moreover all commands and events need to be non-blocking which leads an even more complex state machine.

2.1.2.2 Contiki

Contiki was conceived at the Swedish Institute of Computer Science by Adam Dunkels [Dunkels, Grönvall, and Voigt, 2004]. It came after Tiny OS and introduced an alternative to the purely event-driven concept of Tiny OS: a hybrid model called Protothreads, which are similar to threads on pre-emptive systems like Linux or Windows.

Since its introduction, Contiki gained a lot of attention and has, in addition to its own highly modular network stack, a fully-working IPv6 stack [Durvy et al., 2008] with 6LoWPAN and RPL [Tsiftes, Eriksson, and Dunkels, 2010]. Contiki is the operating system which has been chosen as platform for this master thesis and will be described in depth in Section 2.2

2.1.2.3 RIOT

RIOT [Baccelli et al., 2012] came into existence when a German-French team started analysing the existing operating systems which had already conquered WSNs and Linux as representative of a desktop operating system with support for many programming languages and a comfortable API and came to conclusion that none of those operating systems satisfied their needs.

The basic structure of RIOT is a highly modular and consists of a microkernel that only needs a few hundred bytes of RAM. The kernel itself is written in ANSI C and supports full multithreading, application programming is allowed in C++. Moreover the POSIX standard is in part supported, which makes it even easier for developers to switch to RIOT as a development platform. Another unique feature of RIOT are its real-time characteristics.

RIOT seems very promising and has an active community which also works together with Contiki to ensure interoperability between protocol implementations.

```
PROCESS(protothread_example, "A short description");
PROCESS_THREAD(protothread_example, ev, data) {
  PROCESS_BEGIN();
  //program code here
  PROCESS_END();
}
```

Listing 2.1: An example on how to create a Protothread in Contiki.

2.2 Contiki

As Contiki and its MAC protocols have been chosen as basis of this master thesis, we now provide a short introduction to the operating system and its programming. Contiki has its own programming style due to its design.

2.2.1 Kernel

Contiki's kernel, like Tiny OS is lightweight and event driven. The kernel has two different ways to execute programs:

- Events which are forwarded to processes and their respective event handlers. Once an event handler started execution it cannot be preempted unless it does so itself. There are synchronous or instantaneous events, which are always executed immediately and asynchronous events which are enqueued to be executed later.
- Polling handlers which are used in between asynchronous events to check hardware states.

2.2.2 Timers

There are different timers provided by Contiki in order to enable timed execution in a process:

- `etimer` or event timer, triggers an event on time-out.
- `ctimer` or callback timer, executes a function on time-out.
- `rtimer` is the most precise timer. Also can be used to execute a function on time-out.

2.2.3 Protothreads

Protothreads are like normal threads in programming, but they don't have stack and won't be pre-empted unless the user explicitly does so. This leads to the necessity of global variables for program data and a program state which introduces a bit of complexity compared to unconstrained multi-threaded programming.

Other than that protothreads are very comfortable and easy to use. An example on how to write a protothread is provided in Listing 2.1

Within protothreads execution can be handed over to another thread by calling any of the macros mentioned below, all of which pre-empt the thread:

- `PROCESS_WAIT_EVENT();` wakes up the thread up at the next event posted to it.

- `PROCESS_WAIT_EVENT_UNTIL(ev == ANY_EVENT);` wakes the thread up when the condition inside the parenthesis are met, in this case the event posted is equal to `ANY_EVENT`.
- `PROCESS_YIELD();` hands back the execution after the other thread(s) have handed back execution.

Events can be posted to other protothreads by calling `process_post(&protothread_example, ANY_EVENT , (void*)some_data);`

2.2.4 Energest

Energest is a tool to estimate the energy consumption of a portion of the hardware platform during execution [Dunkels, Osterlind, et al., 2007]. It counts the clock ticks during which a specific hardware component (e.g., radio, LED) is active (e.g., on or off) or in a certain mode (e.g., low-power mode (LPM) for the CPU). While this is very helpful, it still requires knowledge of how much current the part in question uses. The function call `energest_type_time(ENERGEST_TYPE_CPU);` returns the `rtimer` ticks during which the specified resource, in this case the CPU, has been active. Other important resources are `ENERGEST_TYPE_LPM` which returns the time the CPU has spent in LPM, `ENERGEST_TYPE_TRANSMIT` and `ENERGEST_TYPE_LISTEN` which return how long the radio has been active in sending or receiving mode, respectively.

While this estimation is very useful and provides a cheap alternative to expensive hardware, it is not completely accurate. On Energest's accuracy Hurni et al.[Hurni et al., 2011] have found that due to difference in manufacturing the power consumption of the *same node type* can differ up to 4 % .

2.2.5 Low Level Network Stack

Contiki's Network stack is configured at compile time via `#defines` and built in the following way:

- `NETSTACK_CONF_RADIO` is used to specify the hardware module running the radio. In our case that would be the `cc2420_driver`, for the sensortag `cc26xx_rf_driver`
- `NETSTACK_CONF_FRAMER` specifies the frame used for transmission. We normally use `framer_802154`, which is the 802.15.4 frame.
- `NETSTACK_CONF_RDC` is used to specify which radio duty-cycling protocol is used. Examples would be `contikimac_driver` for ContikiMAC or `cxmac_driver` for Contiki's XMAC implementation.
- `NETSTACK_CONF_MAC` is used to specify which MAC protocol is used. Examples: `csma_driver` for CSMA or `nullmac_driver` for no additional MAC protocol.

2.2.6 Network Stacks

Contiki supports two different network stacks: Rime and IPv6. Rime was first developed for Contiki as a lightweight alternative to the IP stack for WSN with much thinner layers, which only provide one functionality at a time. IPv6 is currently on its slow way to conquer the internet and is used by IoT through 6LoWPAN.

2.2.6.1 Rime

The common theme of network stacks in WSN operating systems used to be cross layer communication, resulting in more complex and efficient code [Dunkels, Österlind, and He, 2007]. Adding new applications often was not that easy as knowledge of the underlying layers was mandatory in order to use them

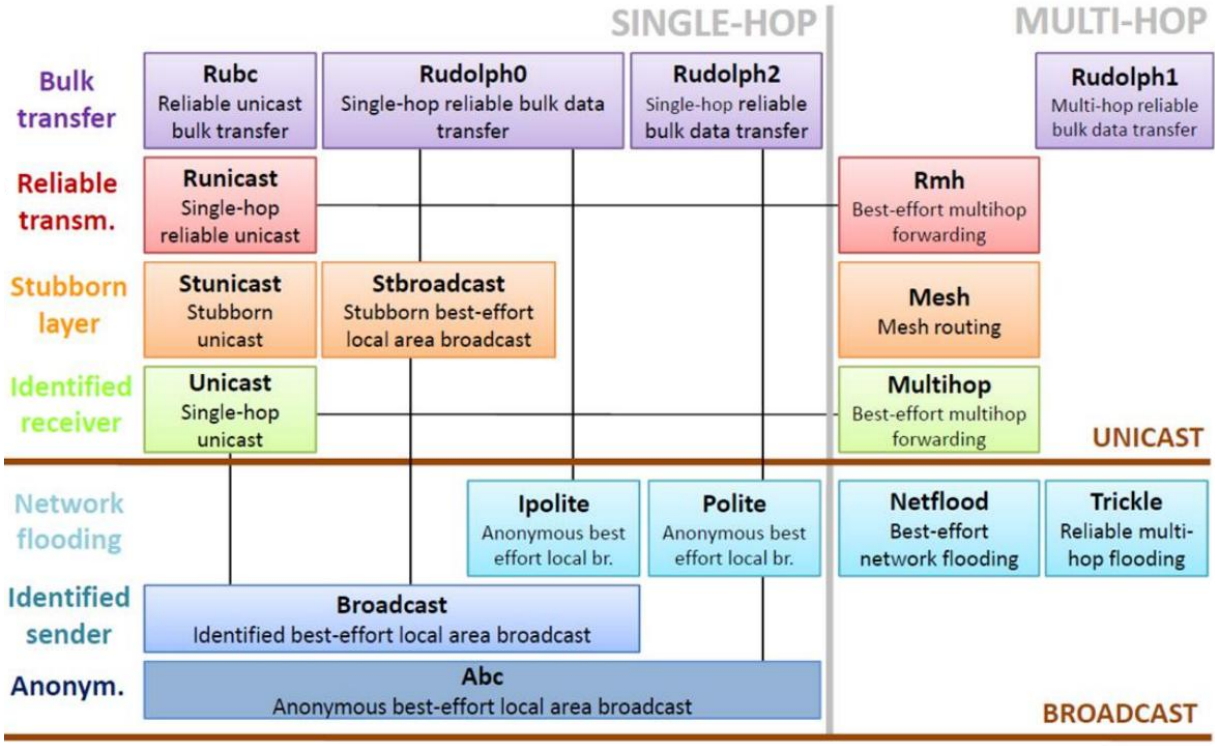

Figure 2.3: Rime's many layers. [Image extracted from Boano [2015c, page 31]]

efficiently. This violates the layering principle, which caused several attempts to create another efficient approach which honoured layering.

Dunkels et al. [Dunkels, Österlind, and He, 2007] came up with a model that had a similar form to IP: The hourglass model, slim at the waist, variety at the ends. Many protocols use header-compression and need to be aligned. For efficiency reasons upper layers decided how the header had to look and then had to also begin to fill those headers. The new approach suggested instead of using the header itself, rather use attributes, which are easy to handle and do not need to be aligned. This could be done in the application layer without any deeper knowledge of the underlying implementation. The translation from header attributes to the header itself are done by Chameleon which according to the configured technology takes care of putting the header together.

Rime itself is lying above this translation module and is a thinly layered collection of protocol primitives, each adding a tiny bit of functionality, starting from anonymous broadcasts (abc) up to reliable-multihop (rmh). The application chooses the kind of connection it needs to maintain with other nodes and also, similar to UDP/TCP has logical channels to separate different communications. While very efficient the rise of IPv6, RPL and 6LoWPAN led to Contiki now being delivered with its IPv6 network stack if not specified differently.

2.2.6.2 IPv6

IP is one of the most successful protocols used in the Internet, still being used after 25 years and it looks like it is going to take several years until the transition to its successor IPv6 is going to happen [Vasseur and Dunkels, 2010]. IP (or IPv4) has a too little address space which lead to the development of the IPv6 standard as early as 1998 [Deering, 1998]. Although many different techniques, like Network Address Translation and private networks have saved IP from the thread of having no IP addresses anymore.

IPv6 is one of the enabling technologies of the IoT: Everything is on the internet. With its 128 bit addresses it is possible to have 3.4 $10^3$8 different addresses or approximately 4.8 $10^2$3 addresses per

person. This is the reason why the dominant IoT operating systems (RIOT and Contiki) have IPv6 stacks enabled by default. IPv6 originally wasn't build for embedded devices but rather for routers which benefit from the design of the IPv6 header: it is always the same length and uses extended headers when needed (see figure 2.3). This format allows several hardware optimizations for routers which do neither run on battery nor have constraint RAM or computing power.

Sensor nodes in WSNs have to act as sensor and as router for its neighbours which makes it necessary to understand IPv6. As sensor nodes do not have hardware optimizations and need to keep packet lengths much shorter than is required by IPv6, it was necessary to find a way to compress the IPv6 header as much as possible, which is done by 6LoWPAN.

2.3 MAC Protocols

Medium Access Control(MAC) protocols are applied on the link layer, controlling access to the radio, deciding when to communicate with the neighbour [Huang et al., 2013], [Roemer, 2014]. There are many different simple mechanisms which can be used to make them more efficient or reliable. Due to its control of the radio, which uses the most power of all the components of the sensor node, it has the most influence on power consumption. Moreover, as MAC protocols are responsible for how to directly communicate it has the most options to secure communications against interference.

2.3.1 Duty-Cycling

Turning the radio on is costly, it doesn't matter if it is to listen or to transmit, the needed power is about the same. For the Tmote Sky as for all the sensor nodes most of the power consumption comes from the RF module (see section 2.1.1.1). Therefore a technique called *Duty Cycling* was developed which allowed the radio to stay in off-mode most of the time and wake up fast in order to receive or transmit a packet.

Doing so introduces new problems: When do devices wake up, how does a device know there is a signal on the air which is indeed a packet and not noise. How to deal with this problem will be explained in the following sections.

2.3.2 CSMA vs. TDMA

The two biggest categories used to classify MAC protocols are CSMA and TDMA protocols. CSMA stands for: Carrier Sense Multiple Access while TDMA means Time Division Multiple Access. CSMA is used to determine whether the channel is busy or not, and is also used to decide whether there is a packet on the air or not. The value measured is the Received Signal Strength Indicator (RSSI) which is normally written in dBm. A threshold is configured which determines above which signal strength the channel is busy. This value should be well above the noise threshold. If in a given area the noise strength and the signal strength are well known, one can choose a good value, slightly below the signal strength.

Carrier Sense also means that sensor nodes which want to send a packet first listen and decide whether it is safe to send. Sensor nodes cannot determine whether a packet has been received successfully, but it can make sure that it started to send when no other signal was on the air which would have been strong enough to make the transmission unusable, or in other words, would have corrupted the transmission. The process of checking if the channel is clear to send is called Clear Channel Assessment (CCA).

TDMA assigns every sensor node its own timeslot to send. No other participant of the network is allowed to send during this time, which should guarantee a successful transmission, assuming there is no noise strong enough to corrupt the packet.

TDMA needs good time synchronization in order to work, which results in an overhead of communication, as clocks will drift apart without readjustments. CSMA will sometimes result in having to try

several times before being able to send, delaying the packet, but not wasting energy. If TDMA is used at all it used together with CSMA, as it normally costs little to nothing but results in much saved energy.
Examples for CSMA: ContikiMAC, S-MAC, T-MAC
Examples for TDMA: LMAC

2.3.3 Multichannel vs. Singlechannel

The physical layer of wireless communication technologies normally provides several different frequency ranges, called channels which can be used to have concurrent transmissions, not interfering with each other. For example IEEE 802.15.4 can operate in three ISM bands (800MHz, 900MHz, 2.4GHz). Lately more of the lower bands have been added to the 802.15.4 bands: 314-316 MHz and 430-434 MHz. In total there are 16 channels in the 2.4GHz band which are usable. But most of them are shared with WiFi, which has stronger signals and can corrupts packets on the air when sent concurrently. Other interference sources which fall into the same frequency band are, for example, Bluetooth and microwave ovens.

Single-channel, as the name suggests, operates on only one channel, and never switches channels. If there is interference or noise which disturbs communication, the underlying MAC protocol has to find a way to deal with that, for example by periodically checking the channel and delaying. Another solution would be that the routing protocol has to abandon that route and find another way.

Multichannel protocols operate on several different channels and change channels on a regular basis. This is useful if there are several different sources of interference on different channels, so that one cannot choose a good and safe channel for communication. This technique is also called channel hopping and requires, like TDMA, good synchronization, and mechanisms to recover if neighbours have not heard from each other for a longer period. Multichannel protocols are much simpler than their counterpart and therefore often preferred if the channel used is guaranteed to be free. Several famous protocols, like Bluetooth Low Energy (BLE) [Gomez, Oller, and Paradells, 2012] and the rest of the Bluetooth family, use channel hopping to be more robust and reliable.

2.3.4 Sender vs. Receiver initiated

Communication can either be started or initiated by the sender or the receiver. For example if the sender wants to transmit its data to another node it starts sending preambles before it starts sending. The receiver then wakes up and starts receiving. No further setup is needed: The sender just starts sending whenever it needs to.

Receiver initiated communication is not used that much in WSNs as it needs further set-up and therefore more messages until the transmission starts. The receiver first sends a preamble, signalling it is ready for data. The sender wakes up and receives the preamble. Afterwards it can start to send its data to the receiver. This is useful if the network is master-slave based and the master needs data from its slave.
Examples for sender initiated MAC protocols: B-MAC, WiseMAC, ContikiMAC
Examples for receiver initiated MAC protocols: A-MAC, Y-MAC, PW-MAC

2.3.5 Flooding (Capture Effect)

The capture effect describes the phenomenon that if two signals within similar signal strength are received, only the stronger one is actually received. This is used by flooding protocols in which all nodes try to transmit the same packet at nearly the same time, which results in a boosted signal. Every node receiving, does not notice the weaker signals, only the one from the transmitter nearest to it, due to the capture effect. Examples are Glossy and Chaos.

2.4 MAC Protocols in Contiki

We now examine in detail the MAC protocols for which an open-source Contiki implementation exists. ContikiMAC has several different MAC protocols and currently more protocols are developed for it, such as Glossy and TSCH.

2.4.1 Medium Access Control vs. Radio Duty Cycling

In Contiki there is a distinction between MAC and RDC. Those two mechanisms, while being on the same OSI layer, are handled separately. Technically, RDC handles the radio access more directly and is used to control when the radio is switched on and off in order to save energy. This in turn impacts the ability to correctly receive packets. CSMA is used as a retransmission mechanism if sending on the RDC layer fails, which is somewhat redundant for RDC protocols with inbuilt retransmissions. Newer protocols do not differentiate between MAC and RDC and handle both mechanisms in one protocol.

2.4.2 NullMAC/NullRDC

NullMAC and NullRDC are the very basic variant of any MAC protocol. No CSMA is used and the radio is always on. This is normally not useful as keeping the radio always on will keep the battery way too fast. The decision to not use CSMA on the other hand can sometimes end up in better results if the interference on the air is so strong and frequent that every CCA attempt fails because the medium is assumed busy. The only alternative for NullMAC is CSMA in Contiki, while NullRDC has several alternatives, which will be described in the following sections.

2.4.3 XMAC

XMAC's primary design idea was to introduce asynchronous duty cycling, as most protocols at this time had at best a combination of low-power listening and synchronous duty cycling [Buettner, Gary V Yee, et al., 2006]. Moreover, long preambles were being used to indicate the imminent transmission of a packet. As written above, the radio module accounts for the highest power consumption and therefore must be kept off as much as possible.

The need to reduce the usage of the radio, gave birth to the idea of shorter preambles, which, to reduce overhearing, also contained the ID of the target. Overhearing means that a node which the packet was not meant for also wakes up and receives the packet, only to discard it afterwards. By being able to decide whether a packet was meant for a node by only receiving the preamble reduces the time the radio has to be kept on, as nodes which were not the intended receivers can go back to sleep immediately. The short preamble, with pauses between each transmission is called a strobed preamble.

If a node receives its ID in a packet it replies with a so-called early ACK which causes the sending node to send the data packet. After the first packet was transmitted the receiver keeps its radio on in order to receive any other queued packets meant for it (see Figure 2.4).

2.4.4 ContikiMAC

ContikiMAC is Contiki's default MAC protocol and has been published in 2011 [Dunkels, 2011]. It is a duty cycling protocol that is sender-initiated. The idea is that the receiver periodically wakes up to execute two CCAs in quick succession and based on that decide whether it is getting an transmission or not. The sender sends its packet as long as it does not receive an ACK (see Figure 2.5). For that to work properly several timing constraints have to be met (see Figure 2.6):

- t_a, time between packet reception and the start of ACK transmission

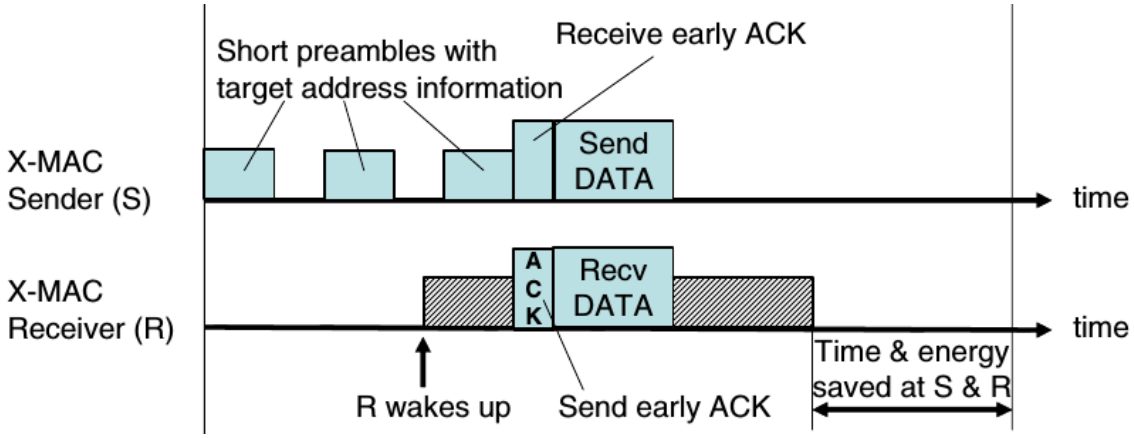

Figure 2.4: The sender transmits small preambles which contain the ID of the receiver. The receiver wakes up and sends back an ACK which triggers the actual transmission. After the received packet the receiver keeps its radio on to receive other queued packets. [Image extracted from Buettner, Gary V Yee, et al. [2006, page 310]]

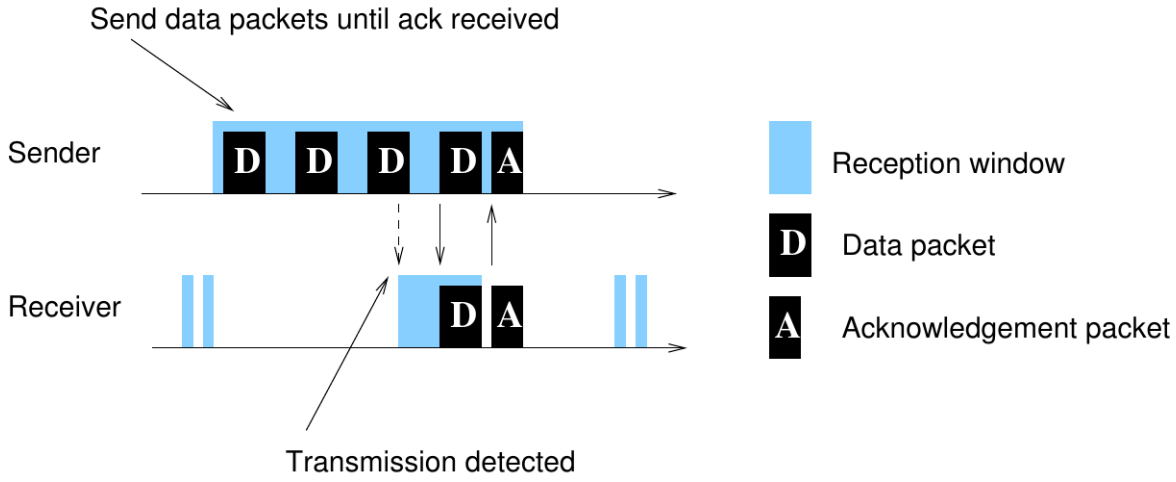

Figure 2.5: The receiver sleeps most of the time and does two CCAs in a short period of time to determine if there is an ongoing transmission. If it received one it stays awake and send back an ACK. [Image extracted from Dunkels [2011, page 2]]

- t_d, amount of time needed to receive ACK
- t_i, the time between to data packets, has to be greater than $t_a + t_d$, otherwise receiving an ACK cannot be guaranteed.
- t_r, the time which is needed to execute a CCA
- t_c, the time between two quick CCAs, has to be greater than t_i otherwise reliable recognition of incoming data packets cannot be guaranteed.
- t_s, minimum packet transmission time, needs to be bigger than $t_r + t_c + t_r$, otherwise packets could fall into the time between two CCAs.

Additionally, to minimize packet retransmission senders can note when the receivers ACK arrived, and use that knowledge to send exactly twice. This is called *phase optimization*.

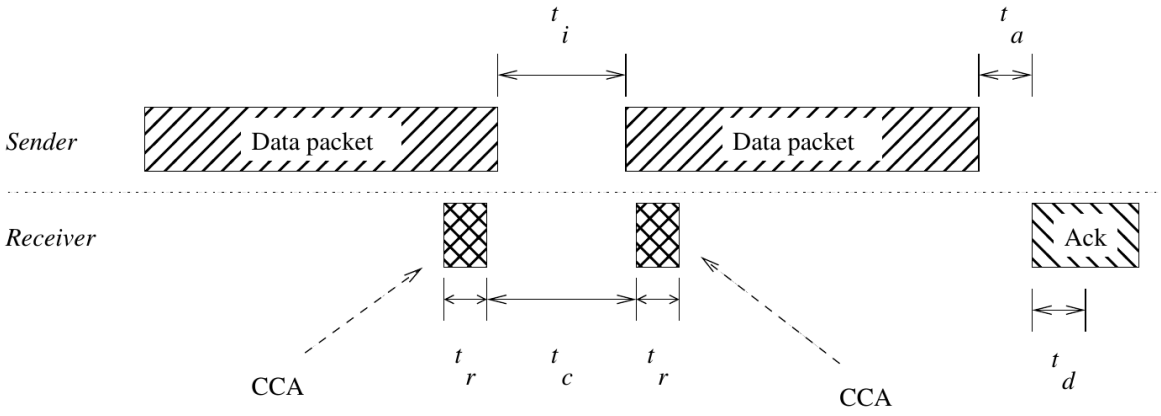

Figure 2.6: The different important timings of ContikiMAC are displayed here. [Image extracted from Dunkels [2011, page 2]]

2.5 FEC

Forward Error Correction (FEC) codes are used to recover corrupted bits in a packet after it has been received. There are properties which are shared by FEC codes: Given a message of k bits and an encoded message with n bits, where $n > k$ and $n, k \in \mathbb{N}$, then at most $\lfloor \frac{n-k}{2} \rfloor$ bits can be recovered, which means for MAC protocols that additional robustness against interference comes at the price of additional bits which have to be transmitted. There are several different FEC codes, but in this thesis only Reed-Solomon Codes are used, which is why only they are introduced in the background section.

2.5.1 Reed-Solomon Codes

Reed-Solomon codes were developed in 1960 by IS Reed and have been used in several different fields [Guruswami, 2010], [Richardson, Iain and Riley, Martyn, 2016]:

- Storage devices such as CDs, DVDs, tape
- ADSL, xDSL
- Satellite communications

Reed-Solomon Codes are defined for integers $1 \leq k < n$, over a Field $\mathbb{F}$ with size $\geq n$ and a set $S = \alpha_1, \ldots, \alpha_n$ as

$$Reed - Solomon_{\mathbb{F},S}[n,k] = \{(p(\alpha_1), p(\alpha_2), \ldots, p(\alpha_n)) \in \mathbb{F}^n | p \in \mathbb{F}[X] \text{is a polynomial of degree} \leq k-1\} \tag{2.1}$$

To encode a message $m = (m_0, m_1, \ldots, m_n) \in \mathbb{F}^k$ the message is interpreted as polynomial

$$p(X) = m_0 + m1X + \cdots + m_{k-1}X^{k-1} \in \mathbb{F}[X] \tag{2.2}$$

The polynomial is evaluated at the points $\alpha_1, \alpha_2, \ldots, \alpha_n$ to get the corresponding codeword. This means that the message-vector is multiplied with the generator-matrix G containing all generator polynomials.

For decoding the message is evaluated at the points $\alpha_1, \ldots, \alpha_{n-k}$ and has to vanish in all those points. If so, the added code can be taken away from the message without further computation as the received message is $c = m + e$.

If less than $\lfloor \frac{n-k}{2} \rfloor$ errors occurred one has to:

1. Find an error locator polynomial (Berlekamp-Massey, Euclid)

2. Find the roots of this polynomials

and then calculate the Symbol error values using the Forney algorithm.

Chapter 3

Experimenting with Interference

In order to understand the influence of interference on the performance of ContikiMAC experiments with repeatable interference were designed. First the environment in which the experiments are carried out and the experimental design are described in Section 3.1. The repeatable interference generator used in the experiments is described in Section 3.1.1. Following are the metrics which shall be determined by the experiments are introduced in Section 3.2. Afterwards the results of the experiments are discussed in Section 3.3. During the experiments issues occurred which are described in Section 3.4.

3.1 Experimental Setup

The testbed used for the experiments is located the Institute of Technical Informatics, Technical University of Graz. It contains up to 90 Tmote Sky sensor nodes distributed over two rooms, Lab North and Lab South. The distribution and location of the nodes can be seen in Figure 3.1. Lab North has a higher number of nodes and higher node density, while Lab South has nodes equipped with SMA antennas, which results in a 6dbm gain and a stronger signal. The final experiments in this work were carried out in Lab South as it has been available more often than Lab North, which was used by other researchers as well.

In Lab South the following sensor nodes were used:

Channel	TX	TX Power	CCA [dbm]	RX	TX Power	CCA [dbm]	Interferer	TX Power
20	205	7	-80	201	7	-80	213	31
11	200	7	-83	211	7	-83	202	31
26	204	15	-83	214	15	-83	212	31

Data from all these nodes was gathered for the experiments. The results of the nodes was then averaged and the standard deviation calculated. Every experiment consisted of a certain amount of packets, mostly 1000, being sent from a dedicated sender node to its assigned receiver. During the whole duration of the experiment the same interference level is present, in order to find out how well a protocol does under certain levels of interference. After the experiments the results were evaluated by the metrics given in the section below. The interference for the experiments was created using a dedicated interfering node, running JamLab, explained in Section 3.1.1.

Every experiment was redone many times during the development phase. For the evaluation one experiment with 1000 packets to be sent, was run during the night, when the external interference should be minimal.

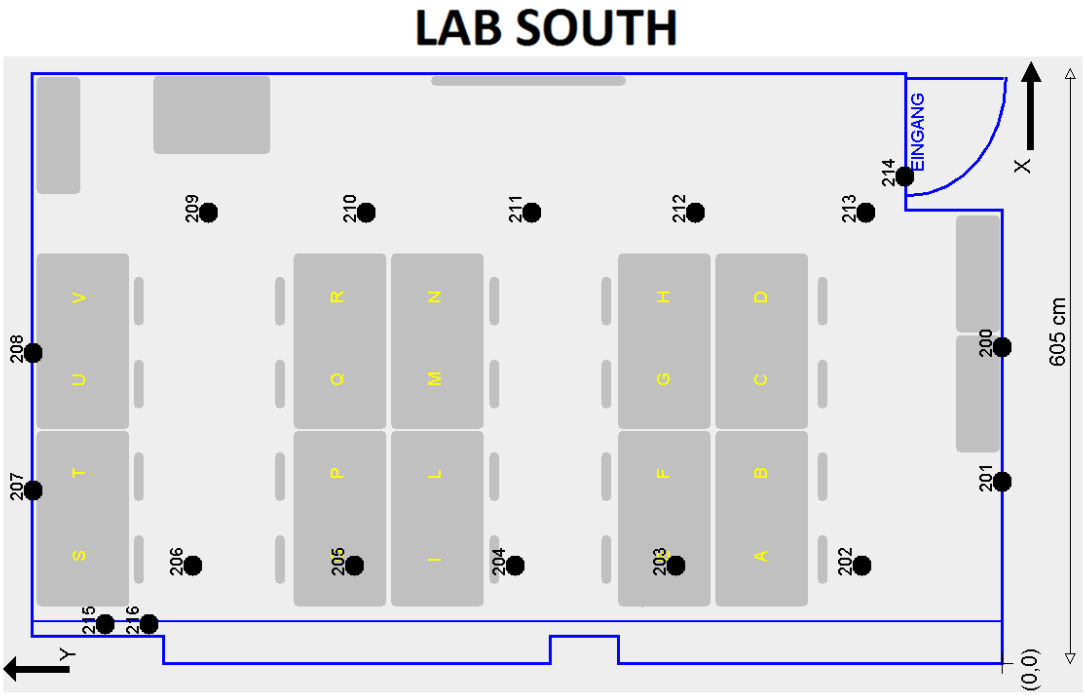

Figure 3.1: In this picture the locations of all the nodes in Lab South can be seen. [Image extracted from Boano [2016]]

3.1.1 JamLab

While the community was developing protocols it was lacking the means to debug and test them efficiently. It was hard to create interference reliably and neigh impossible to do so in a repeatable manner. In 2011 JamLab [Boano, Voigt, et al., 2011] was developed as a tool to aid developers when creating a protocol. Testbeds at universities were normally equipped with many nodes but not with dedicated interfering devices, which were configurable over the internet. JamLab solves that problem by turning ordinary sensor nodes into interfering devices. This also means that no additional hardware needs to be bought in order to create interference. JamLab's target platform is the Tmote Sky and the CC2420 radio.

In order to create repeatable interference JamLab uses prerecorded interference patterns which is replayed at runtime. Such patterns contain video-streaming, surfing or data transmissions over WiFi, microwave ovens, which when switched on interfere with a wide range of channels, or Bluetooth, which due to its channel hopping also affects a wide variety of frequencies.

In order to record interference a much higher sampling frequency of the medium is needed than in normal 802.15.4 operations. This is because WiFi sends at a higher frequency than 802.15.4. Therefore the cpu speed needs to be boosted and can reliable catch all 802.11b communications. Unfortunately with the CC2420 it is not possible to reliably detect the smaller 802.11g/n frames, which means we cannot emulate this kind of interference.

This tool enables developers to test their protocols in the same settings and therefore enables meaningful performance comparison between protocols. This is why JamLab was decided as this Thesis' generator of interference.

As mentioned above JamLab incorporates many different styles of interference, from audio streaming via 802.11b to total jamming of the carrier. In our experiments we only use the different WiFi interferences as these are the most commonly found.

- WiFi1: is audio streaming and very little interference, but today many households use audio streaming.
- WiFi2, is video streaming which is slightly stronger than audio streaming. In all the experiments only WiFi2 is used representing weaker interference.
- WiFi3, is FTP file transfers, which nearly jams the carrier. Here we can see how well protocols fare against harsh conditions.

3.2 Metrics

The metrics which are used to determine how well a protocol does are:

- Success Ratio: how many packets which are sent by the application, arrive successfully at the target.The success ratio tells us how reliable a protocol is, a crucial point for all protocols. For some applications a low success ratio might be sufficient as it only contains non-critical data, such as a fridge communicating its content, or a coffee-machine sending its tank status. Also for audio-streaming application where realtime is more important than receiving all packets a low success ratio might be sufficient. For other applications such as a security system or health care products losing packets might not be acceptable.
- Total Energy consumption: energy consumption in general is an important characteristic for MAC protocols. Remote sensor nodes need to use as little battery as possible and therefore communication must use as little resources as possible and not communicating should not cost battery. As we will notice the PRR heavily influences the energy consumption.

- Latency: the time between sending the packet and the successful reception. While this is an important characteristic for some applications, this work will not put any emphasis on latency.

3.2.1 Secondary Characteristics

All of the aforementioned metrics, can be calculated by gathering data on the MAC layer and below, and monitoring the energy consumption. As mentioned above there are different kinds of events that will happen during communication under interference, which will be used to determine the metrics.

- Number of successfully received packets,
- number of acknowledged packets,
- number of sent packet,
- number of packets dropped due to channel activity, referred to as deferred, packets
- number of corrupted packets,
- number of lost packets, which were sent but not noticed at the receiver. These can only be calculated off-line,
- number of lost acknowledgements, which were sent by the receiver but did not reach the sender. These as well can only be calculated off-line, after a finished experiment.

Some of the secondary characteristics do not directly relate to the metrics but might indicate why a protocol does poorly and what can be done as a countermeasure. Moreover, knowing exactly what happened to a packet increases certainty in the data.

In order to gather this data, we need to look at the device driver and the MAC protocols, which contain the low level part of the communication. We need to count every event and print updates on what is happening during the experiment.

There is such a mechanism in Contiki which has been used to determine how well Rime and its protocols do, but we are not interested in Rime and its many layers, but MAC protocols such as ContikiMAC. RimeStats gives a good idea on how to implement collecting the data and actually gathers a good amount of what we will need, but as will be seen later, some data is still missing.

Concerning the energy consumption, energest will be used to estimate the energy used during communication.

All this data, RimeStats and energest, will be printed over the serial interface, with every node printing its data over its own device, in addition to some additional output by the protocol. This data will be parsed afterwards by post-processing.

The data which is needed to calculate this metrics is gathered using RimeStats, an inbuilt data gathering tool which counts certain events listed below. All of the meanings given to the items were extracted directly from the code. If no further explanation was given to an item, it was not used for any calculation or in any protocol experimented with.

- **tx**, contains every successfully sent packet,
- **rx**, contains every successfully received packet,
- **reliabletx**, contains every packet which is not a broadcast and passed NullRDCs CCA before sending,
- **reliablerx**, not used,

- **rexmit**, not used,
- **acktx**, not used,
- **noacktx**, not used,
- **ackrx**, contains the acks the sender received
- **timedout**, not used,
- **badackrx**, not used,
- **toolong**, form of corruption on the driver layer,
- **tooshort**, form of corruption on the driver layer,
- **badsynch**, form of corruption on the driver layer,
- **badcrc**, form of corruption on the driver layer,
- **contentiondrop**, contains the packets which failed the CCA in the driver
- **sendingdrop**, not used,
- **lltx**, contains every packet loaded into the sending buffer,
- **llrx**, contains every packet received on the driver which was not corrupt in any. way

The metrics which were used were calculated in the following way:

$$\text{Experiment Packet Number} = \text{Number of Packets enqueued to be sent} = 1000 \tag{3.1}$$

$$\text{sent} = \text{lltx}_{sender} - \text{contentiondrop}_{sender} \tag{3.2}$$

$$\text{corrupt} = \text{toolong}_{receiver} + \text{tooshort}_{receiver} + \text{badsynch}_{receiver} + \text{badcrc}_{receiver} \tag{3.3}$$

$$\text{received} = \text{llrx}_{receiver} + corrupt \tag{3.4}$$

$$\text{PRR } \% = \frac{\text{llrx}_{receiver}}{sent} \tag{3.5}$$

$$\text{Lost } \% = \frac{sent - received}{sent} \tag{3.6}$$

$$\text{Corrupt } \% = \frac{corrupt}{\text{llrx}_{receiver} + corrupt} \tag{3.7}$$

$$\text{Sender Success Rate } \% = \frac{\text{ackrx}_{sender}}{ExperimentPacketNumber} \tag{3.8}$$

$$\text{Deferred } \% = \frac{\text{contentiondrop}_{sender}}{\text{lltx}_{sender}} \tag{3.9}$$

$$\text{Acks Lost } \% = 1 - \frac{\text{ackrx}_{sender}}{sent} \tag{3.10}$$

$$\text{Success Rate } \% = \frac{\text{llrx}_{receiver}}{\text{Experiment Packet Number}} \tag{3.11}$$

How reliable the protocol is can be seen through the **Success Rate**, which describes the portion of packets which arrived successfully at the application layer. The sender success rate shows how reliably the sender achieved to send a packet and receive an ACK, basically the reliability of the connection from the sender's point of view.

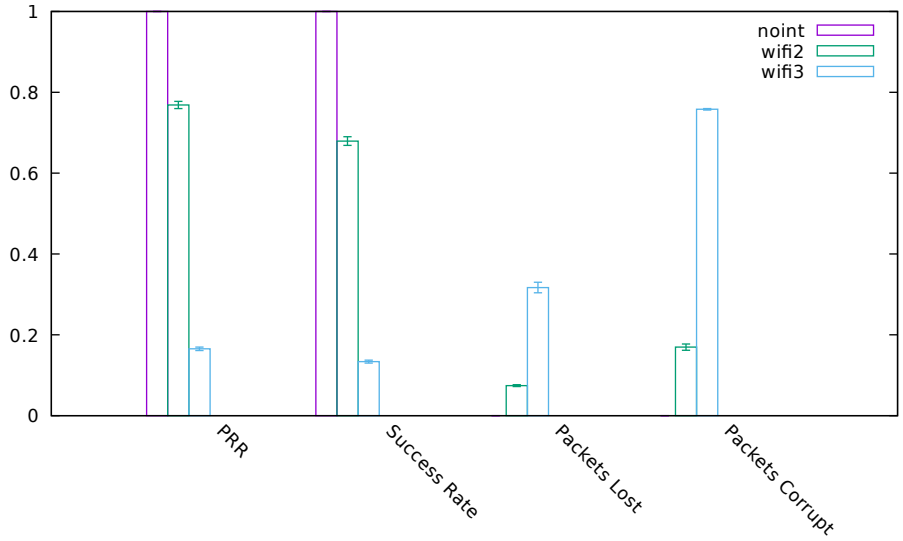

Figure 3.2: Metrics calculated from the data gathered for NullRDC

3.3 Results

The results of the first experiments under interference are documented here. First the whole setup was verified with NullRDC, first with no interference, then with WiFi2 (video streaming) and WiFi3 interference (FTP). Afterwards ContikiMAC was tested in order to understand how ContikiMAC behaved under interference, and also why, as this understanding is necessary for improving the protocol.

3.3.1 Verification: Aloha (NullRDC)

In order to have some comparability between ContikiMAC and NullRDC considering the timing and on how long the experiments take, the delay between each packet sent for NullRDC was $\frac{1}{8}$ of a second. We can see in figure 3.2 that NullRDC does not do too bad performance wise, but under heavy interference more than 40% of the packets are corrupt. The PRR is slightly higher than the Success Rate or PDR, as it is only counts packets which were really sent. In other words: If a packet is sent, how high is the probability for it to arrive at the receiver correctly. Not every packet which is queued to send is actually send. Some are dropped or deferred if the CCA check directly before sending is not passed. We can see that up to 15% of packets are not sent due to heavy interference in figure 3.3. As NullRDC does not have any form of retransmission, every deferred packet will not get transmitted to the receiver.

3.3.2 ContikiMAC

For ContikiMAC the stats indicate that most packets, which are sent, are lost, even without interference, which can be seen in figure 3.4. This is the reason why the PRR is below 50%, but this has no influence on the reliability itself, but the energy consumption. A lower PRR means more retransmission and there more energy consumption. Considering that some packets will always be lost if the sender has to find the receivers schedule, the result for no interference is good. The success rate tells us how reliable the protocol all in all is and for no interference it is 100%.

Upon further investigation we notice that `lltx` does not describe every low level packet sent, but rather every packet that is loaded into the buffer, which is a big difference in ContikiMAC. This means we will need more data in order to correctly calculate the packets which are actually sent in ContikiMAC,

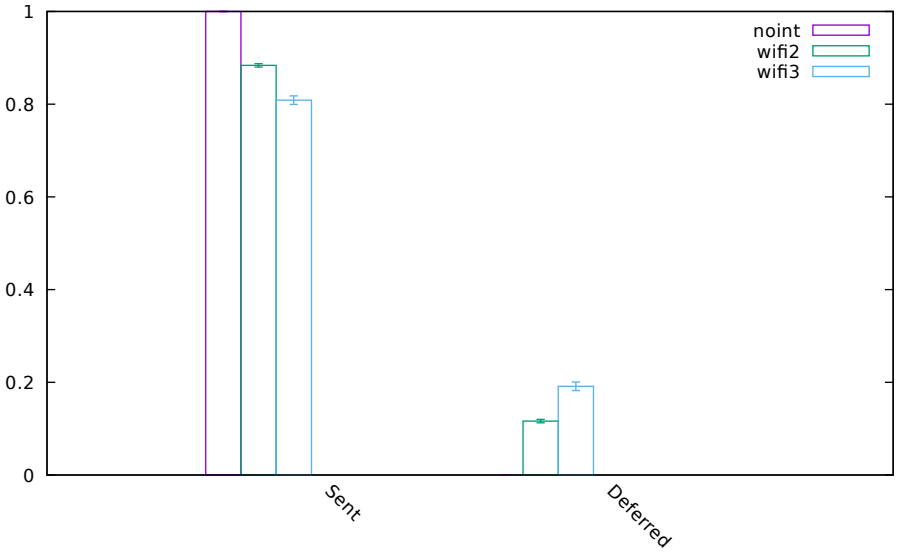

Figure 3.3: Metrics calculated from the data gathered for NullRDC

and also to find out why ContikiMAC does so poorly under interference. Although we can determine the Success Rate correctly we do not know why exactly ContikiMAC crumbles under interference

3.4 Issues during Experimentation

Currently a tool provided by Contiki called serial-dump. It can read from one or more serial devices at the same time, in a single thread. It can send commands to the nodes, but every node will receive the same command, which often does not make sense. The data of all nodes read, will be written into one file, which can then be parsed later.

This is not ideal for many reasons:

- To read many nodes at once one can start one process for each node which has to be logged, or write one's own tool.
- Sending commands to single nodes during the experiment is not possible. Only global commands are possible if one instance of serial-dump is started for all nodes and with the approach of one process per node sending individual commands is possible but not practicable.
- Configuration is therefore not possible on an individual level.

It would be much better to have:

- One thread per node inside a single process.
- A possibility to configure every single node individually, reducing the number of resets and flashes.
- Possibility to have control during the experiment, for example: `print current status`.
- Do some precomputations in order to reduce effort during the post processing phase.

Not being able to configure also introduces another issue: Every reconfiguration means recompilation and then re-flashing. Depending on the testbed structure reflashing equates to having to wait longer than

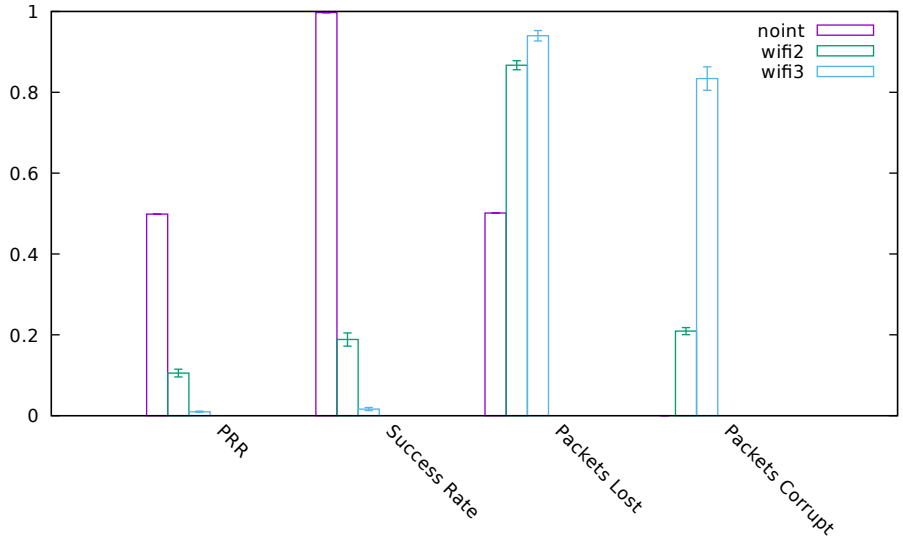

Figure 3.4: We can see that without interference performance is good, but even when weak interference is present the performance drops significantly. The data itself does not suggest a clear culprit, so further investigation is necessary.

the experiment might take, especially during development where most of the time different approaches are tried out.

The issues which occurred during experimentation together with the data which is needed to accurately analyse ContikiMAC led to the decision to build the framework, described in Chapter 4.

Chapter 4

Framework

In the last chapter problems that arose during experiments were described, as well as data which was missing. In this chapter first the motivation for implementing a framework assisting in executing and evaluating experiments will be given in Section 4.1. The complete design and reasoning behind it are given in Section 4.3. After that the implementation and all its details are described in Section 4.4. Finally in Section 4.5 some improvement opportunities of the current implementation are discussed.

4.1 Motivation

In order to ensure easy evaluation and analysis of protocols a test framework was written. The first step was to gain a deeper understanding of Contiki's protocols under interference which was not possible with current tools available in Contiki. How many packets were actually sent and passed the CCA? Why was this transmission aborted? In order to design countermeasures against interference a deep knowledge of what happens during a transmission is needed. This was the leading design ideal.

Another design ideal was to ensure smooth experiments with a clear, easy and automatable workflow. As during development and fine-tuning of a protocol many experiments are necessary, the less time is spent on configuration, compilation and re-flashing the faster the progress. This is even more important as often experiments are executed on large testbeds with many nodes, sometimes up to hundreds of nodes, where re-flashing will take even longer.

Generally these experiments all are similar: A large number of packets is sent from sensor node A to sensor node B, sometimes over a multi-hop network, sometimes only over a single hop. For this thesis only the single hop case is observed. During transmission, an interferer is activated to disturb the communication. Now packets are lost or not sent due to channel activity, acknowledgements are lost and packets are corrupted. All of these events have to be recorded in order to be evaluated later.

While this does not sound complex at all, there are three different challenges that arise when doing experiments:

- Generating repeatable interference
- Software running on the nodes
- Software running on the host

4.1.1 Generating Repeatable Interference

The problem of generating repeatable interference was solved using JamLab and described in chapter 3. JamLab itself is not integrated in this framework.

4.1.2 Testing Software running on the nodes

For every single experiment the node configuration has to be adjusted. Concerning Contiki that means:

- the configuration has to be adapted, for example the frequency of wake-ups for duty cycling protocols,
- Contiki has to be recompiled and the sensor node has to be flashed with the new binary

This is quite time consuming and flashing multiple sensor nodes can take a wile, dependant on the infrastructure used for experimentation. Furthermore every node will print debug-output which can be used for evaluation via its serial connection. These outputs, also called traces, are recorded on a server, which is connected to the nodes. After the experiment is finished the traces are analysed and post processed to extract the data sought after and finally a new experiment can be started, after reconfiguration. As many different experiments are necessary in order to analyze, improve, debug and fine-tune a protocol every time saving mechanism should be built into the testing framework. In general the software running on the nodes should apart from being highly configurably online also gather a variety of data which is described in Section 3.2. The complete implementation is described in Section 4.4.2.

4.1.3 Testing software running on the host

All the nodes are normally connected to one or more servers. On it a piece of software is running which collects the serial output of all the nodes. This data is called traces and normally the software doing this job is simple and cannot do anything apart from reading the serial device. Additionally the current implementation is single-threaded which means that in order to collect data from more than one node, a second process needs to be started which makes sending commands, such as configuration parameters or start and stop commands to nodes harder. The implementation of the host is described in Section 4.4.1.

4.2 Requirements

There are several different requirements to a framework:

- Scalability: There are huge testbeds with several hundreds of nodes. The framework should be able to handle a testbed that big. Moreover it should be very easy to add more nodes to an already configured experiment.
- Extendability: It should be fairly easy to add new commands and configuration options to the framework. Often during experiments the need for more, additional data arises, which is why there should be one file in the framework which needs to be changed to add additional data.
- Reduction of flashing the testbed: Every time the need to flash the testbed arises, time is wasted, which is why as much as possible should be configurable during runtime. This also needs to be added to Contiki itself, as many of the configuration options are decided at compile time. This will not be necessary any longer for the latest hardware, as it has much more memory available and could therefore handle being configured at runtime. This, we cannot influence though, we have to work with what is given by Contiki.

4.3 Design

With the requirements mentioned above in mind the following design, see figure 4.1, came to light

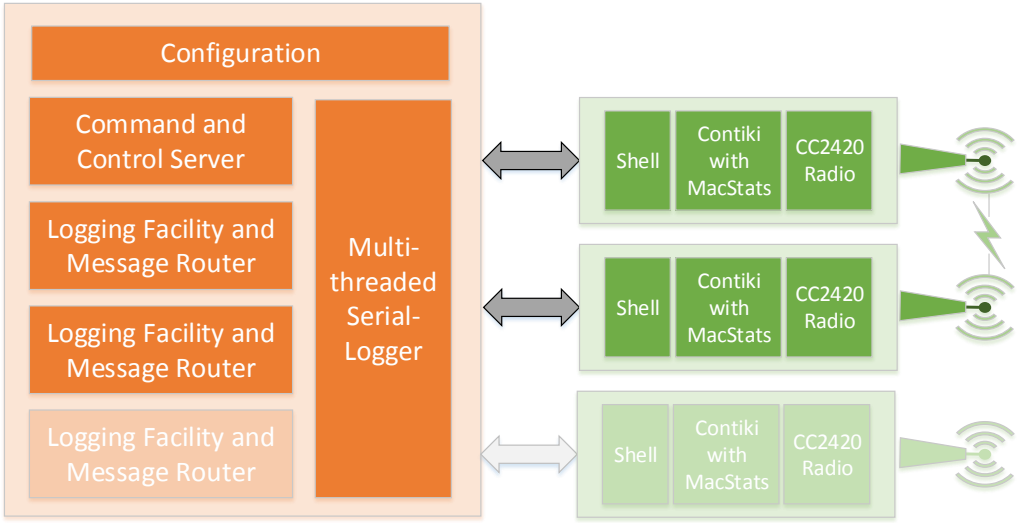

Figure 4.1: On the left hand side the host/server can be seen which reads the configuration and sends it to the nodes at start up. The nodes can be found at the right hand side which communicate via the serial interface with the host. For every node a thread is started which relays commands and logs the output of the nodes.

- Node - Test Client: Every node has the same piece of software on it, running a certain protocol. It waits for instructions of the controller and its configuration before the experiment can be started. As the aim is to test point to point connections every node has to be paired with a communication partner, which is given to node by its configuration. Additionally the CCA threshold, sending power, channel, who is the sender and the receiver, are all transmitted to the node by its configuration.

  After that an experiment can be started by the controller, which consists of many packets being sent from the sender node to the receiver node. When the experiment is done, the nodes signal they are finished to the controller, indicating a new experiment can be started.

  On the node, in the MAC protocols data is gathered. Adding a new datum only requires manipulating the statistics file and adding a line where to count the event. As every MAC protocol has its own metrics and data which influences its performance this is a necessity.

  Adding new commands also is very easy and is taken from the existing shell structure of Contiki, which means little effort is needed to implement new commands.

- Server - Controller/Host: Every node is connected the controller which runs the testing host. At startup it reads a configuration file with the configurations for all the nodes which take part in the experiment. After that connection to the nodes is established, a check is made whether all the nodes are connected, the configuration transmitted and a thread for communication is started.

  Afterwards an interactive shell is started enabling the user to control the nodes and adjust configurations is necessary. Then the experiment can be this started. During the experiment the output of all nodes will be written line by line to the terminal, which should enable users to follow their experiments in real time. For every node, log-files are created containing only their output. If a node's message starts with a `[tag]` a new file only containing those messages is created. New tags can be added easily and new files will be created automatically. This makes it easier to have a fast overview of how the experiment is going and eases post-processing.

```
Rime-Address Channel TX-Power Partner-Address Continuous-Mode[0/1=on] CCA[dBm]
    Sender/Transmitter[0/1=sender]
0.19 26 15 0.18 0 -35 1
0.18 26 15 0.19 0 -35 0
```

Listing 4.1: The first line describes the basic syntax of the config file while the other two lines show an example of two nodes set up for an experiment. The nodes with addresses 0.18 and 0.19 communicate on channel 26 with TX power set to 15 and a CCA threshold of -35 (which equals to -80dBm). 0.19 is the sender and the experiment is not automatically repeated afterwards.

4.4 Implementation

As implementation language C was chosen. C is very efficient, but developing in C comes at a cost: Development is much slower than with a high-level language and it comes with less comprehensive libraries which are not needed anyways. The implementation language for the Contiki program was C as well, which was predetermined by the Contiki platform itself.

4.4.1 The Host

The program is started as follows:
`./test_program -b <baudrate> [-f <savedirname>] -c <config-file> -p <numberofpackets> <serialdevice1> <serialdevice2> [more serial devices] ....`
At start the program reads the config-file, which has several options per node connected to the computer. The config-file has one line per node which contains, in any order, all the nodes which should be used for testing. The syntax per line can be seen in Listing 4.1. The program saves all the logs in the given directory, which, if it did not exist at program start, will be created. The baudrate determines the speed with which the host-pc communicates with the sensor node(s). The sensor nodes' serial devices, which are used for the experiment, are written at the end of the program invocation.

After reading the config-file the host will try to connect to all the serial-devices and send them the configuration after connecting successfully. Then the command-prompt will be printed, the user can make further adjustments to the experimental setup or start the experiment itself, by issuing the start command. See Figure 4.2 for an example on how a program start looks like. Most of the commands are just sent to every node connected, but some commands are also or only affecting the host. If a command is not known by the host, it is automatically sent to all devices. The main commands are:

- `start` - starts an experiment, also sends a command to all devices to start the experiment
- `stop` - stops an experiment, sends stop command to all devices and flushes all files,
- `continue` - continues a stopped experiment, sends continue command to all devices
- `send [node id] command` - send one command to one node
- `save` - all the files currently in use are flushed, nothing is sent to nodes themselves
- `exit` - exits the program, also writes and closes all open files

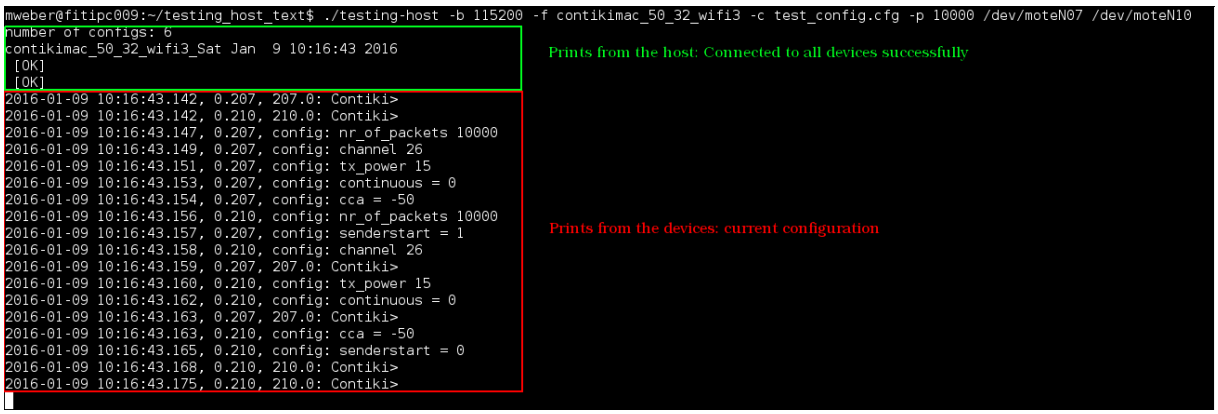

Figure 4.2: The program is started with a baudrate of 115200, a packet number of 10.000 and two nodes connected two the serial devices `/dev/moteN07` and `/dev/moteN10`. After program start the configuration is read and after checking the serial devices for actual nodes running the test-client, which is indicated by `[OK]`, the configuration is sent. The nodes print their configurations and wait for further commands afterwards.

4.4.2 The Node

The nodes have to run the a certain program which features a basic shell, which is provided by Contiki. The nodes understand the following commands:

- `ccathreshold [threshold in dbm]` - sets the CCA threshold of the node.
- `txpower [power from 0 - 31]` - sets the transmission power of the node,
- `rfchannel [11 - 26]` - sets the communication channel of the node.
- `start` - starts an experiment, with the role given by the configuration. It also resets all the recorded statistics, so that if an experiment was run before it does not influence the statistics of the current program.
- `stop` - stops the experiment,
- `continue` - continues the current experiment,
- `getMoteId` - the command used to check if the serial device is a node. The node answers with its ID.
- `test-config <number of packets> <channel> <TX power> <target node id (e.g. 0.19)> [<continuous = y(/n)>]` - this command is used to transmit the configuration in one go to the client.

Adding more commands to the client is, as was required, fairly easy. The code for an additional command can be seen in Listing 4.2. At the moment there are several more commands which come with the basic shell. To view these send the `help` command to the nodes.

After an experiment has been started it runs until completion. At completion the nodes each send a message, indicating that the experiment has been completed. Another experiment can then be started. All messages will be logged in one file, additionally for every line printed starting with a `[tag]` a new file will be generated which contains all the lines that started with `[tag]`. As such adding a new `[tag]` is very easy: It is done by adding a `printf("[newtag] data\n");` to the client. Currently the following tags exist:

```
PROCESS(cmd_process_name, "process_string");
SHELL_COMMAND(command_name,
              "command_string",
              "command_string",
              &cmd_process_name);

PROCESS_THREAD(cmd_process_name, ev, data) {
  PROCESS_BEGIN();
  // command code here
  PROCESS_END();
}

//shell code
PROCESS_THREAD(protocol_testing_shell, ev, data) {
  PROCESS_BEGIN();
  //initialization
  serial_shell_init();

  ...

  //own commands
  shell_register_command(&cmd_name);

  ...

  PROCESS_END();
}
```

Listing 4.2: At first a new process has to be declared. Afterwards a new shell command is created. The code for the command itself is in the `PROCESS_THREAD` and any passed arguments apart from the command itself can be found in data. Finally the command has to be registered in the shell which is running on the client.

- `[energy]` - Which contains the cycles for the cpu in low power and normal mode, TX and RX mode of the RF module. There are several other interesting values, like the clock cycles spent in interrupt routines, but as these are neglectable compared to the energy consumption of the RF module they are not used.
- `[macstats]` - This tag contains the statistics which are being logged, for example: packets sent, packets received, packets lost, packets corrupted, and many more. All of the statistics taken are printed, but not always all of them are actually useful for every protocol. The decision to always print all of the statistics came with the desire to be able to always use the same plotting script for the statistics.
- `[rimestats]` - This tag pronounces the inbuilt statistics which are shipped with Rime. The MAC statistics provide all the Rime statistics plus additional others.
- `[packet]` - These lines contain per packet statistics, like the RSSI and the LQI, as well as the packet number itself. These are only printed by the receiver, on receiving a packet.

4.4.3 MacStats

After the start command has been received, the sender-node starts sending packets. Every 500 packets (this is adjustable) it prints out the current MAC statistics. The MAC statistics are mostly recorded in the CC2420 driver, but some are in the RDC/MAC layer, as there are several statistics which are unique to a protocol, for example duplicate detection or lost ACKs. The following statistics are taken:

- `lltx` - driver level, preparing transmission
- `llrx` - driver level receiver wake-ups
- `lltx_ok` - driver level finished transmission
- `lltt` - driver level transmission calls (included CCA failed transmissions as well)
- `tx` - successfully transmitted packets which means the MAC driver return TX_OK, counted in the application
- `rx` - successfully received packets in the test client
- `corruptrx` - the packet received was corrupted, but passed the MAC layer, happens rarely
- `acklosttx` - the packet sent was not acknowledged
- `toolong` - the packet received in the driver was too long to be a packet
- `tooshort` - the packet received in the driver was too short, maybe just a false wake up
- `badsynch` - the first byte received indicated a packet which is too long
- `badcrc` - the packet did not pass the CRC check. If FEC is available the packet will be corrected if possible
- `collisiontx` - the driver could not send the packet because of a failed CCA check
- `collisiontx2` - the driver could not send the packet because of a failed CCA check, possibility number 2 in the cc2420 driver
- `duplicate` - the MAC driver received a packet it had already received. This probably happened due to a lost ACK

- `collisionlosttx` - the packet was not sent and the network stack returned that the packet could not be sent. This happens after several tries of sending the packet and failing the CCA check, meaning the medium was busy every time an attempt to send was started.
- `ackcollisionoverheardtx` - ContikiMAC did get noise instead of an ACK and therefore stopped sending.
- `contikimacdeferred` - ContikiMAC found a busy channel before starting to strobe and therefore dropped the packet.
- `colrxwtx` - receiving a packet just before sending, in ContikiMAC, counted as collision
- `acklostcomptx` - the sender could not make sure its packet was sent due to missing ACKs and the network stack returns it lost the packet due to an absence of an ACK. This is rather impropable as ACKs are very short and the probability is higher to lose the packet sent.
- `unknownlosttx` - if the network stack returns it lost the packet without a reason it would be logged here
- `badcrclost` - only counted if FEC is on and the packet had too many errors to be corrected
- `badcrcrecovered` - only counted if FEC is on and the packet was recovered
- `onlycrcrecoverd` - if the packet passed the crc check but FEC
- `swackrx` - software acks received
- `swackfectx` - software acks send, only used within ContikiMAC with FEC

It is fairly easy to add additional statistics. In the statistics.h file there is a struct containing all the statistics taken. In there the new value is added. After that it has to be added to the print function as well, which has due to efficiency reasons, all of the statistics printed hardcoded. At a point it was done by a loop which was so inefficient, that it was slowing down the experiments. Adding a value to the can be done by calling `MACSTATS_ADD(value_name);`.

Every second the energy used is printed for the following node modules:

- RF Module:
  - TX: The cycles the RF Module has been on and in transmission mode
  - RX: The cycles the RF Module has been on and in listen mode
- CPU:
  - LPM: The cylces the cpu has been running in low power mode
  - normal: the cycles the cpu has been running in normal mode

4.5 Discussion

At first it was planned to create a host which did most of the experiments automated and on its own. Taking C for that undertaking was not the optimal choice. Any other language with true multi-threading and a vast library would have been better.

- One of the ideas was, in order to speed up communications and to omit `printf` on the node side, to implement binary communication, or in other words: to send the data objects directly. The problem that arose was that this would have cost much time, due to serial driver adjustments which would have been necessary. Also debugging on the nodes would have been harder, as `printf`-debugging would not have been possible, without adding extra modes in the driver. As the target was to implement the test-client without interfering too much with Contiki's underlying code-base, so it would be possible to adopt it in the repository, this idea was then dropped.

- In the beginning the experiment layout also considered having the two nodes communicating with each other taking turns in sending, in order to find out if multipath-fading had influenced the performance of the tested protocol. This did not work well under interference, as the nodes had to make sure that every message was delivered, before continuing to send the next test-packet. Otherwise collisions due to the test-protocol were possible, as one node was thinking that it had already successfully received a message, while the other did not receive an ACK so it retried to send an outdated packet.

  The solution, which also resulted in much simpler test-client code, was to run the whole experiment with only one sender. Afterwards, if necessary, a second experiment would be started which switched the initial sender and receiver settings. This adjustment made it unnecessary to make sure that every packet arrived, which meant that if a MAC protocol decided that a packet got lost, the next in line would be sent and the packet was logged as lost without any further added complexity.

- Another issue was the growing complexity of the host. At one point during this project, the host calculated many of the statistics on-line, which made it necessary to implement another command structure in which certain `[tags]` in the logs from serial devices resulted in calculations. This would have been much easier with binary objects, but as this failed due to reasons described before, it was necessary to parse everything. As this is much easier in object-oriented languages or languages with larger library support than C (e.g. Python, C++), this was then done either while plotting or with extra python scripts.

- An issue which we found in the version of Contiki we used was a bug in Energest which already existed in previous versions but was not discovered until another bug in the clock of the MSP430 was fixed. The change in behaviour of Contiki was found with git-bisect. The problem at hand was that the numbers of two exclusive, but always present modes (Low Power Mode and Normal Mode of the CPU) did not add up. Energest lost cycles with every switch between these two exclusive modes, as well as direct switches from RX to TX and vice versa of the RF module. The basic problem was that switching from one mode to another was done with two separate macros which led to a loss of cycles in between. Instead the new macro `ENERGEST_SWITCH(type_off, type_on)` (for the full code see Listing 4.3) which had to be used instead of the separate `ENERGEST_ON(type_on)` and `ENERGEST_OFF(type_off)` macros. Mr. Atis Elsts kindly provided us with a bug-fix which was then applied to Contiki and led to the desired behaviour.

```
#define ENERGEST_SWITCH(type_off, type_on) do { \
    rtimer_clock_t now = RTIMER_NOW();                                    \
    if(energest_current_mode[type_off] != 0) {                            \
      energest_total_time[type_off].current += (rtimer_clock_t)(now -     \
          energest_current_time[type_off]);                               \
      energest_current_mode[type_off] = 0;                                \
    }                                                                     \
    energest_current_time[type_on] = now;                                 \
    energest_current_mode[type_on] = 1;                                   \
  } while(0)
```

Listing 4.3: Instead of two separate macros only one macro has to be called to switch directly from one state to another. While this still is not perfect, it loses unnoticably little cycles and is therefore much better and far more accurate than its predecessor.

Chapter 5

Revisiting ContikiMAC

After gathering new data via MacStats, ContikiMAC is now ready to be re-evaluated with new insights. Therefore at first the metric calculation is updated in Section 5.1. Afterwards in Section 5.2 the earlier experiments are repeated for ContikiMAC in order to gain new insights on its actual performance. The results are discussed in Section 5.3 where improvements to Contiki's performance are suggested. The experiments are repeated with these improvements applied and its results shown in Section 5.4.

5.1 Metric Updates

Due to the new data, the metrics needed to be updated. In order to prevent confusion: ContikiMAC does not necessarily stop to send due to failed CCA checks, but there are several stages in the protocol where a packet can be dropped due to noise on the channel, which is why there is a difference between collisions, which simply result in a packet not being sent by the driver, giving control back to the MAC protocol, and dropped packets, which ContikiMAC decided it could not transmit.

The following statistics have been used for the plots:

$$\text{sent} = \text{lltx\_ok}_{sender} \tag{5.1}$$

$$\text{collisions} = \text{collisiontx}_{sender} + \text{collisiontx2}_{sender} \tag{5.2}$$

$$\text{dropped} = \text{contikimacdeferred}_{sender} + \text{colrxwtx}_{sender} + \text{ackcollisiontx}_{sender} \tag{5.3}$$

$$\text{corrupt} = \text{toolong}_{receiver} + \text{tooshort}_{receiver} + \text{badsynch}_{receiver} + \text{badcrc}_{receiver} \tag{5.4}$$

$$\text{received} = \text{llrx}_{receiver} + corrupt \tag{5.5}$$

$$\text{PRR } \% = \frac{\text{rx}_{receiver}}{sent} \tag{5.6}$$

$$\text{Lost } \% = \frac{sent - received}{sent} \tag{5.7}$$

$$\text{Corrupt } \% = \frac{corrupt}{received} \tag{5.8}$$

$$\text{Sender Success Rate } \% = \frac{\text{tx}_{sender}}{sent} \tag{5.9}$$

$$\text{Dropped } \% = \frac{dropped}{\text{Experiment Packet Number}} \tag{5.10}$$

$$\text{Acks Lost } \% = \frac{\text{rx}_{receiver} - \text{tx}_{sender}}{\text{rx}_{receiver}} \tag{5.11}$$

$$\text{Success Rate } \% = \frac{\text{rx}_{receiver}}{\text{Experiment Packet Number}} \tag{5.12}$$

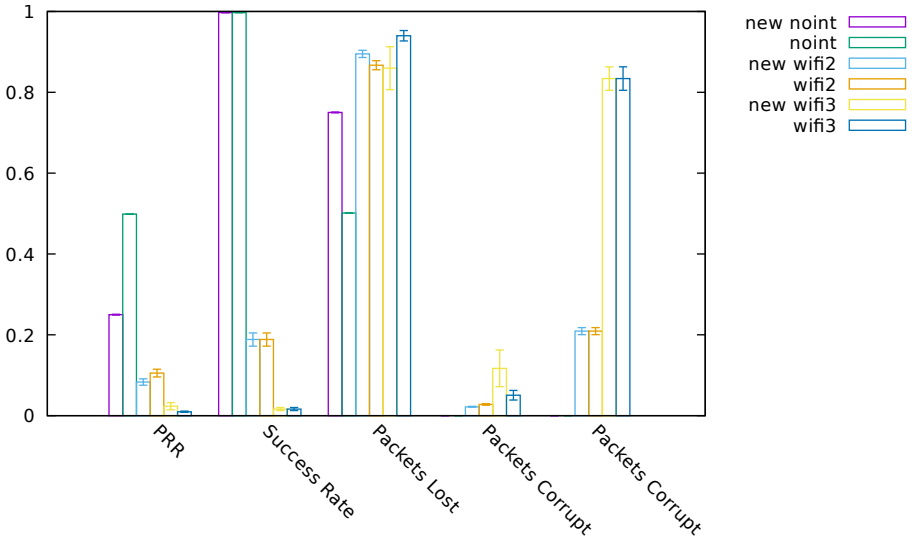

Figure 5.1: Results of the former experiment compared to the results gained from MacStats.

5.2 Results

In order to see the difference between these experiments and the ones done before they are shown side by side in Figure 5.1. This time we used the correct amounts of packets which were actually sent, meaning on the air, passing the CCA. As we saw before the PRR is not that great and the real amount of transmissions done by ContikiMAC does even makes it worse, but this accounts only for protocol analysis and energy consumption.

Earlier only the amount of packets loaded into buffer was counted as transmitted packets. This had an influence on all the stats calculated apart from the Success Rate, which only looks at how many of the enqueued packets were delivered, which is why it did not change. This also means that concerning the reliability of ContikiMAC no new insights were gained.

It can be seen that for heavy interference most packets are lost, which can be explained to the receiver being asleep when the actual sending took place. But is that really what happened? If we look at how many packets were dropped early in Figure 5.2, we find out that three quarters of the enqueued packets were dropped before the nodes were even trying to send them. Another fifth was early dropped due to noise which happened to be on the channel instead of an ACK. For a node running ContikiMAC which retransmits for one period until it receives an ACK this is a fatal flaw under interference.

5.3 Discussion

From these results can be seen that ContikiMAC suffers very hard under interference, the problem being not even starting to strobe, but instead return `MAC_COLLISION_TX` to the upper layer. If there would be a CSMA layer on top of the duty cycling protocol, that would make it slightly better, but if the interference stays the same for a longer period of time even that will not change much.

The exact part in the code where this decision is taken can be seen in Listing 5.1. The channel is repeatedly checked for activity. If any is found, the transmission will be cancelled, and not even begun. Before that in the code is a check for incoming transmissions, so this is redundant, which is why it was disabled. This made it also necessary to disable the Phase Lock mechanism as it did not work any more with the changed timing. We leave it to future work to fix that mechanism, as it also introduces other problems as well.

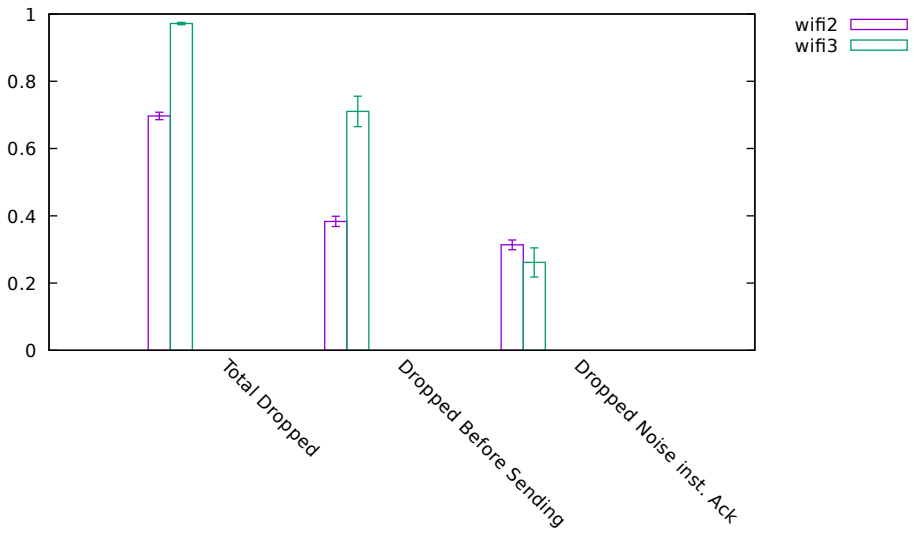

Figure 5.2: Packets dropped by ContkiMAC for various reasons. If there is no interference, no packets are dropped, which is why there is no interference test results included in this figure.

```
  for(i = 0; i < CCA_COUNT_MAX_TX; ++i) {
    t0 = RTIMER_NOW();
    on();
#if CCA_CHECK_TIME > 0
    while(RTIMER_CLOCK_LT(RTIMER_NOW(), t0 + CCA_CHECK_TIME)) { }
#endif
    if(NETSTACK_RADIO.channel_clear() == 0) {
      collisions++;
      MACSTATS_ADD(contikimacdeferred);
      off();
      break;
    }
    off();
    t0 = RTIMER_NOW();
    while(RTIMER_CLOCK_LT(RTIMER_NOW(), t0 + CCA_SLEEP_TIME)) { }
  }
}

if(collisions > 0) {
  we_are_sending = 0;
  off();
  PRINTF("contikimac: collisions before sending\n");
  contikimac_is_on = contikimac_was_on;
  return MAC_TX_COLLISION;
}
```

Listing 5.1: For several times the medium is checked for activity. If one of those CCA checks fails the transmission is stopped. The default configuration for the amount of CCAs is 6.

```
NETSTACK_RADIO.transmit(transmit_len);
// ...
wt = RTIMER_NOW();
rtimer_clock_t st = wt;
while(RTIMER_CLOCK_LT(RTIMER_NOW(), wt + INTER_PACKET_INTERVAL)) { }

if(!is_broadcast && (NETSTACK_RADIO.receiving_packet() ||
    NETSTACK_RADIO.pending_packet() ||
    NETSTACK_RADIO.channel_clear() == 0)) {
  uint8_t ackbuf[ACK_LEN];
  wt = RTIMER_NOW();
  while(RTIMER_CLOCK_LT(RTIMER_NOW(), wt + AFTER_ACK_DETECTECT_WAIT_TIME)) { }

  len = NETSTACK_RADIO.read(ackbuf, ACK_LEN);
  if(len == ACK_LEN && seqno == ackbuf[ACK_LEN - 1]) {
    got_strobe_ack = 1;
#if WITH_PHASE_OPTIMIZATION
    encounter_time = txtime;
#endif
    st = RTIMER_NOW() - st;
    printf("got ack after %d strobes, time %d\n", strobes, st);
    break;
  } else {
    printf("contikimac: collisions while sending\n");
    collisions++;
  }
}
```

Listing 5.2: If during the strobe process, after sending a packet instead of an ACK, channel activity is found strobing is stopped altogether

The other problem mentioned before was stopping the strobes earlier than one full period if instead of an ACK the channel was busy. In a noisy environment this will happen more often than not. The code which led to that behaviour and was therefore disabled, can be found in Listing 5.2.

5.4 Revaluation

After disabling those two optimisations mentioned before, the experiment is repeated. In Figure 5.3 it can be observed that instantly the Success Rate is increased by far if any interference was present. For FTP interference nearly 50% of enqueued packets were delivered, up from about a percent or two. Another notable fact is that around 85% of all the packets received are corrupt. Maybe there is a way to reduce the number of corrupt packets and increase the Success Rate.

Unfortunately the newly found performance comes at a cost: The energy consumption for the transmitter increases heavily, indicated by the lower PRR, which can be seen in Figure 5.4, but being more reliable will always come at a cost.

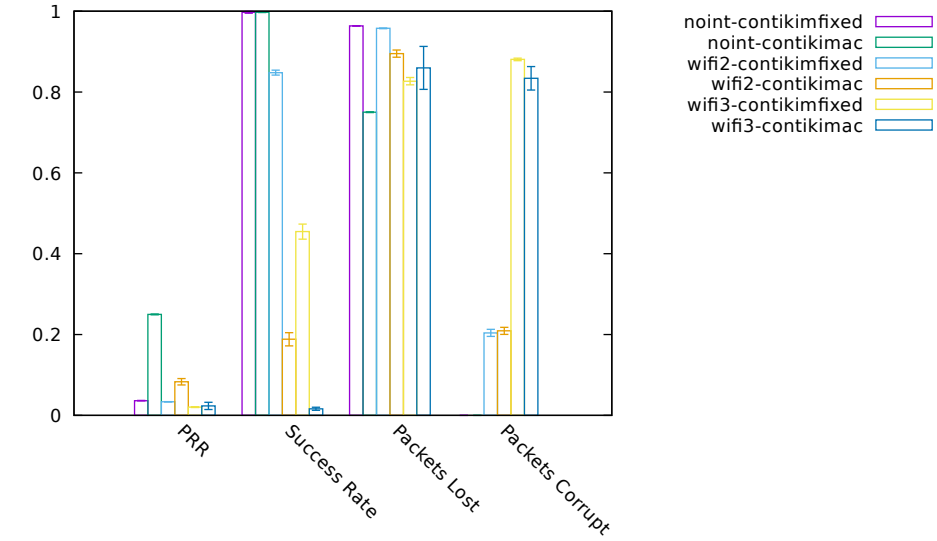

Figure 5.3: Stats for the improved ContikiMAC

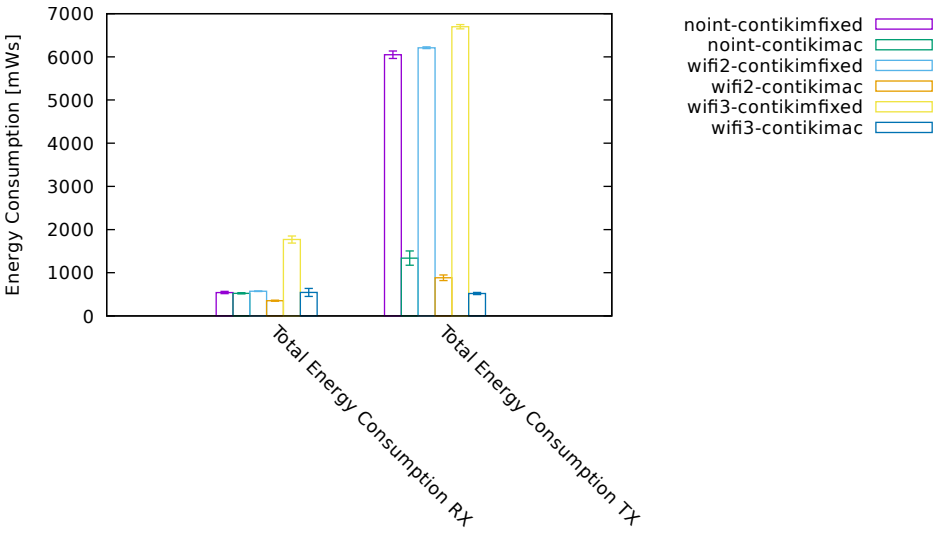

Figure 5.4: Total Energy Consumption for the improved ContikiMAC

Chapter 6

ContikiMAC with Forward Error Correction (FEC)

In the last chapter it was shown that ContikiMAC suffers from many corrupt packets, which arrive at the receiver instead of meaningful communication. Due to this it was decided to employ FEC as is used in BuzzBuzz, described in [Liang et al., 2010]. First, in section 6.1, in this chapter is the initial design described which is used together with the existing ContikiMAC design. After that the addition of FEC into Contiki is described in Section 6.2. In Section 6.3 the experiments which were done in the previous chapters are repeated for ContikiMAC with FEC, and its results are presented there. Finally in Section 6.4 the results are discussed and further improvements suggested.

6.1 Protocol Design

Basically adding FEC to ContikiMAC, is adding the code itself after the data and, should the CRC check fail, call correction function to rectify any incorrect bytes. Unfortunately it is not that easy:

- FEC needs a fixed packet size which has to be transmitted, which means the size of the packets is pre-determined. This can be omitted but then extra information has to be transmitted with the frame: How many bytes of error-code have been added to the packet.

  For every packet size there seems to be an amount of error-code that improves the overall performance, which means that it is its own work to find a good method for handling smaller packets. The decision was take to the same approach as BuzzBuzz and take 50 bytes of data together with 15 bytes of 802.15.4 frame and encode them with 30 bytes of error-code.

- FEC is slow. In [Liang et al., 2010] runtimes from 100ms up to 200ms depending on how many errors occurred were observed. Our implementation needs 55ms without errors up to 140 if 15 errors had to be corrected. While our implementation is faster it is still far too slow to work with ContikiMAC in its current form, as ContikiMAC waits for an ACK for a short period of time and then sends the next strobe. This time needs to be short as it has to be smaller than the two CCA checks the receiver does to determine if there is a packet to receive.

To solve that problem the following steps were taken:

1. Sender sends packet with FEC.
2. As soon as a packet arrives, and it is recognized as such an early ACK is sent.
3. On arrival the sender stops sending further strobes and switches its radio off, waiting for the receiver to perform FEC on the sent packet.

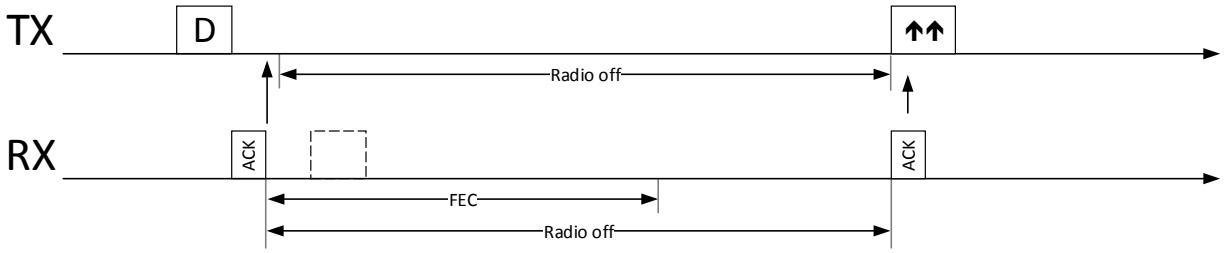

Figure 6.1: Design of ContikiMAC: In this case the early ACK arrives and the sender stops sending and waits for the rest of the strobe period and the duration of the correction for the ACK.

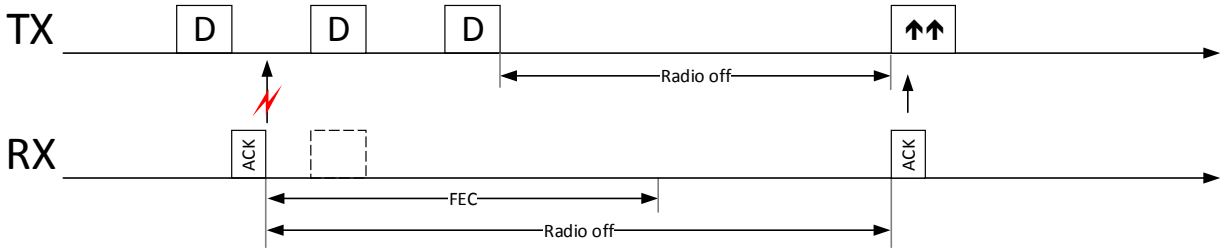

Figure 6.2: Design of ContikiMAC: In this case the early ACK does not arrive and the sender keeps sending for the whole strobe period and waits afterwards for the duration of the correction for the ACK.

4. The receiver also switches off the radio and corrects the packet. It then waits until the maximum time which could be needed for FEC has passed and sends another ACK in order to signal that the packet was correct.

5. The sender wakes up after the $t_{FEC,max}$ and waits for the ACK. Afterwards the status is returned.

This still suffers from a serious problem: What if the first ACK is lost? As the automatic hardware ACKs of the CC2420 cannot be used. Hardware ACKs are only sent if the CRC check passes, which means for the early ACK in the other cases software ACKs have to be used. Software ACKs are slower and have therefore a different timing than hardware ACKs which is why only software ACKs were used.

The solution for this problem results in further information which needs to be sent to the receiver: The last two bytes of each packet sent now contain the time which had passed since the sender started to send strobes. This information needs to be updated with each strobe, which makes it necessary to reload the buffer with the timing information. This also makes it impossible to encode those two bytes with FEC, as it takes too long. This is why the timing information bytes are located after the error-code.

With this information the receiver can calculate when the sender would stop to strobe if the early ACK did not arrive. The receiver then waits, after correcting the packet, for the rest of the strobe duration and the $t_{FEC,max}$ until it responds with an ACK. This takes a long time which is why no higher channel check rate than 4 is possible with FEC. This design is illustrated in Figure 6.1 and Figure 6.2

6.2 Implementation

ContikiMAC previously did not have any FEC implementation included which is why the decision was taken to include an Reed-Solomon Code implementation with ContkiMAC. In order to make it easier to use for others who want to enhance any protocol with FEC it the calls to encode and decode any packet were included in the packet buffer and are ready to use. The function's names are:

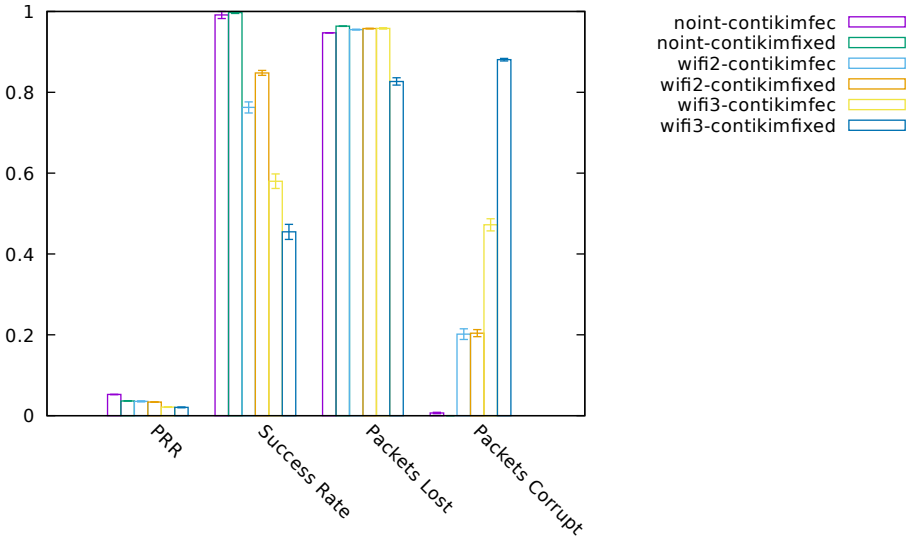

Figure 6.3: Stats for ContikiMAC with FEC.

- `int packetbuf_encode_rs();` after the framer was called, or the header has been generated. A negative return value means that encoding has failed, which only happens if the code did not fit into the packet any more.
- `int packetbuf_decode_rs();` before the packet is handed to the framer. A negative return value means that decoding has failed due to too many errors in the packet. If successful, the function return 0 if the packet did not have to be altered, and the number of errors corrected otherwise.
- `void remove_rs();` removes the error code from the packet and hides it from the upper layers.
- `packetbuf_set_corrupt(uint8_t status);` is used to mark corrupt packets in the driver layer. Reflects the status of the hardware CRC check.
- `uint8_t packetbuf_get_corrupt();` is used in the MAC layer to determine whether the driver's CRC check failed or not.

Furthermore a function to send software ACKs was included with ContikiMAC as it is needed there twice. Generally it seems advisable to have one 802.15.4 software ACK implementation for Contiki but at the moment there is no dedicated space for it.

6.3 Experimental Results

The results of the experiment show a massive improvement in reducing corrupt packets for heavy interference, which can be seen in Figure 6.3 , but the timing problems which come through using software ACKs and the long period of waiting for the error correction seems to make things worse otherwise. In Figure 6.4 can be seen that FEC is less effective for weak interference, as less of the corrupt packets can be corrected. In addition to that, many ACKs which are sent from the receiver, are lost, which can be seen in Figure 6.5. The energy consumption of ContikiMAC with FEC can be seen in Figure 6.6 is by far higher, which can be explained by the many missed ACKs and the lower CCR which leads to longer strobing periods.

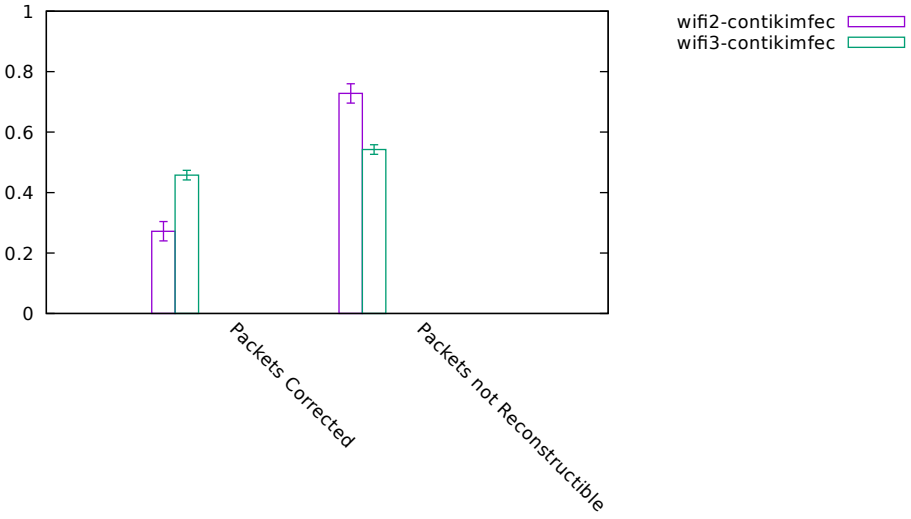

Figure 6.4: Amount of correctable and incorrectable packets.

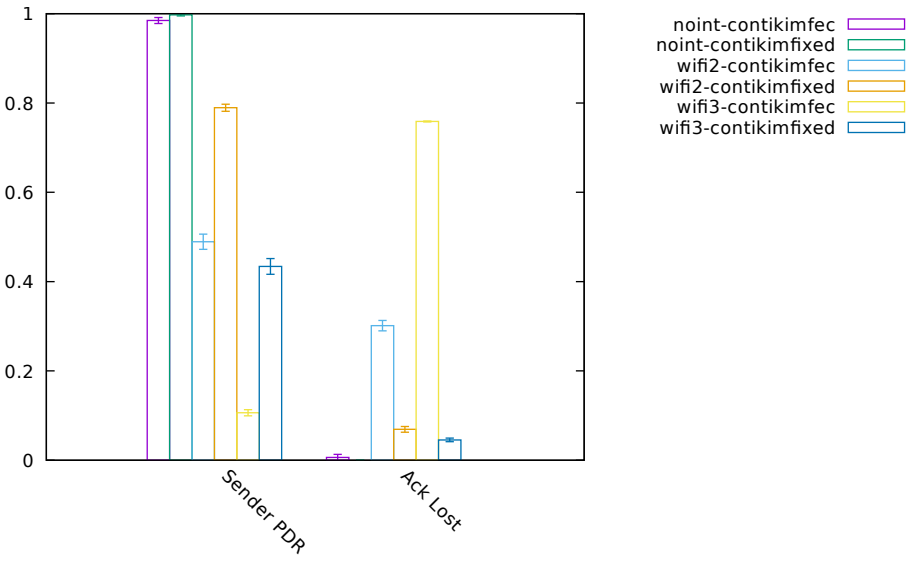

Figure 6.5: Sender Stats for ContikiMAC with FEC.

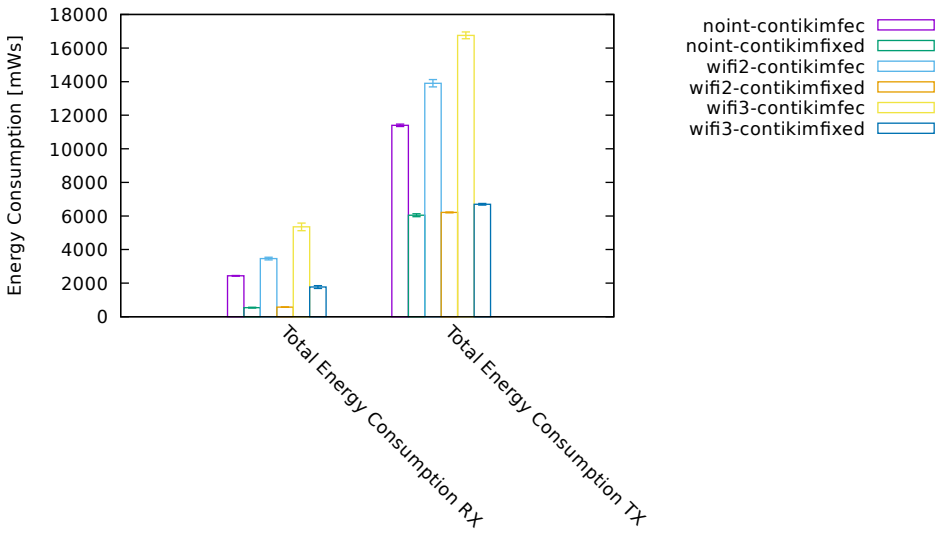

Figure 6.6: Total Energy Consumption of ContikiMAC with FEC compared to the improved ContikiMAC.

6.4 Discussion

We see that the addition of FEC reduced the amount of corrupt packets for heavy interference by nearly 50%. For weak interference the addition FEC seemed to reduce the total number of corrupt packets, but the additional corrupt packets that occurred due to the additional length of the packet seemed to add enough incorrectable packets, to make it not beneficial to use FEC in that case. Due to timing problems we can see that not all the ACKs arrive at the sender, which is something we might look into.

Chapter 7

ContikiMAC with FEC and Jamming Agreements (JAG)

In the last chapter we saw that lost ACKs are a serious flaw of ContikiMAC with FEC which is why in an attempt to improve the amount ACKs received the Jamming Agreement is added to ContikiMAC with FEC which is shown in this chapter. In Section 7.1 the changes to the design are explained, and shown. Afterwards the implementation details can be found in Section 7.2. Following are the results of the experiments in Section 7.3 and finally the discussion of those results in Section 7.4.

7.1 Design

The addition of JAG did not change the design in its core, but it made it easier to receive the second, final ACK. The two scenarios described in chapter 6 stay the same, but the final ACK is replaced with a Jamming sequence. The scenarios can be seen in Figure 7.1 and Figure 7.2. If the early ACK arrives its measured RSSI is used for searching for the jamming sequence. If no early ACK arrives, the RSSI must not return to the noise floor for the duration of the jamming in order to be recognized as such.

7.2 Implementation

The implementation is heavily based on source code provided by Dr. Boano, which was included in my Contiki implementation. The code needed for jamming is based on the platform itself, which is why the necessary code was placed in `platform/sky/dev/`. In order to start a jamming sequence the function `jamming_acknowledgment(rtimer_clock_t duration, uint8_t carrier_type, uint8_t jamming_pow` needs to be called. The parameters are used as follows:

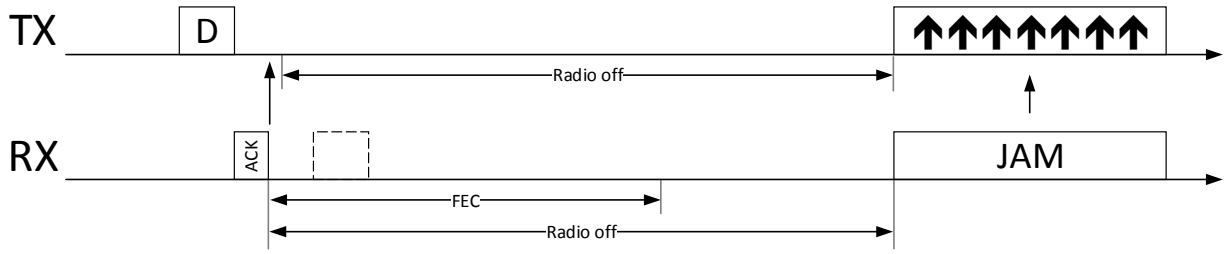

Figure 7.1: Design of ContikiMAC: In this case the early ACK arrives and the sender stops sending and waits for the rest of the strobe period and the duration of the correction for the Jamming Agreement.

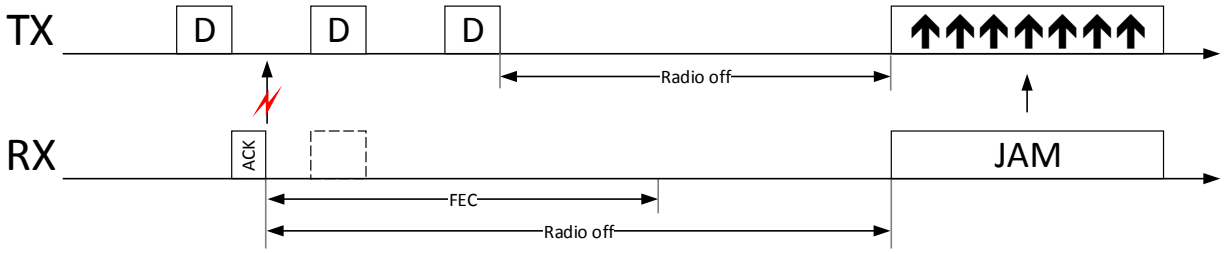

Figure 7.2: Design of ContikiMAC: In this case the early ACK does not arrive and the sender keeps sending for the whole strobe period and waits afterwards for the duration of the correction for the Jamming Agreement.

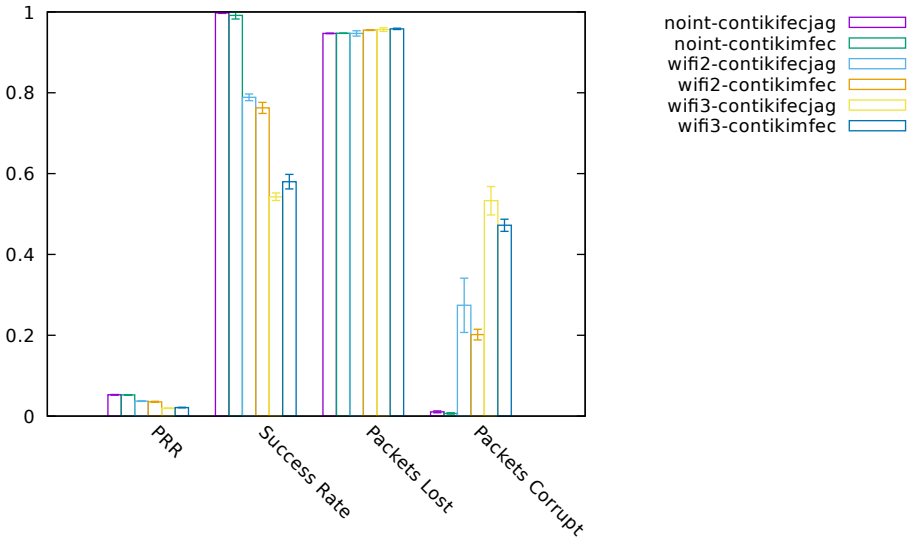

Figure 7.3: Stats for ContikiMAC with FEC and JAG.

- `duration` defines the duration of the jamming sequence.
- `carrier_type` selects the type of jamming. 1 is for a modulated carrier and 0 for an unmodulated one.
- `jamming_power` adjusts the strength of the jamming and is essentially the same as the transmission power.

In order to check for jamming sequences the function
`uint8_t jam3_check_for_jamming(int8_t wanted_rssi);`
has to be called. `wanted_rssi` describes which RSSI is the expected jamming level. If a value below the noise floor is passed then the signal must not return to the noise floor to qualify as jamming.

7.3 Results

We can see in Figure 7.3 that the addition of JAG did not really impact the receiver side performance. Performance under heavy interference decreased slightly, while performance under light interference increased. JAG was only meant to increase the accuracy of information for the sender, which can be seen in Figure 7.4. The energy comsumed with JAG is again higher than its previous version, where only an ACK was sent. This can be seen in Figure 7.5.

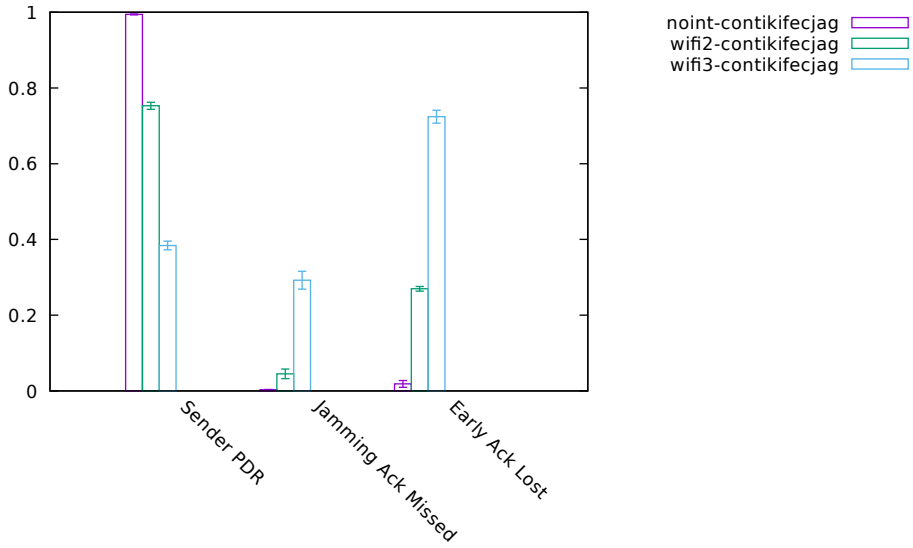

Figure 7.4: Sender Stats for ContikiMAC with FEC.

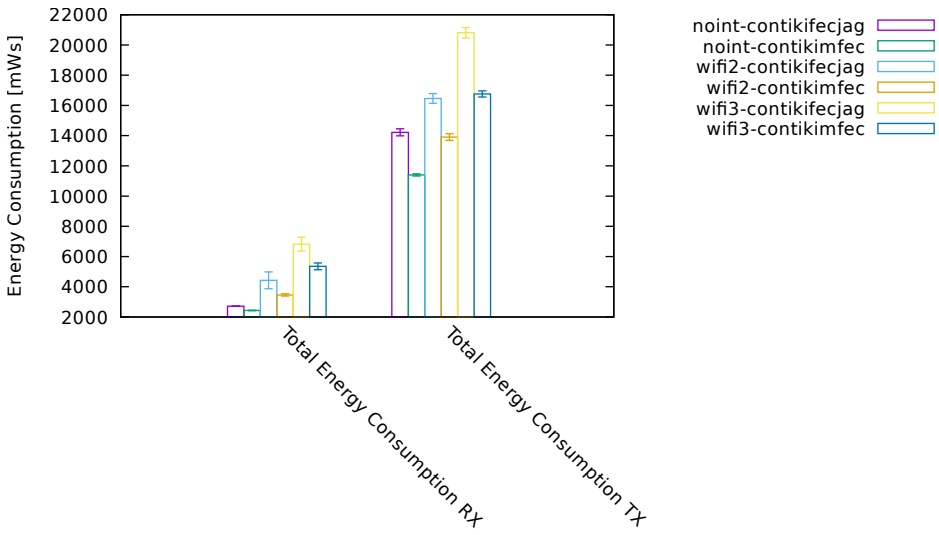

Figure 7.5: Total Energy Consumption of ContikiMAC with FEC and JAG compared to ContikiMAC with FEC and normal ACKs.

7.4 Discussion

The addition of JAG provided us with far better ACK reception than the previous version of ContikiMAC with FEC. There might still be the possibility of further improvement for overall reliability of the protocol by fixing the timing issues it has, as many of the problems this protocols has stem from it being slowed down by FEC.

The high energy cost can partly be explained by the attempt to send as fast as possible. Without Phase Lock this will result in the maximum number of sent data strobes by the sender. This means that in turn for the protocol to work better the Phase Lock mechanism needs to be fixed.

Chapter 8

Outlook and Related Work

Interference has been a problem for WSN for a while, so there have been different approaches and helpful tools in mitigating its effects:

8.1 Forward Error Correction

The idea to use Forward Error Correction codes to enhance the performance of a protocol stems from [Liang et al., 2010] which is used in their protocol BuzzBuzz. The protocol itself uses multiple headers as well as FEC to enhance its performance.

In contrast to this work, they used real networking hardware for generating interference in their experiments and analysed WiFi's (802.11b/g) impact on 802.15.4. In order to make sure that no hardware specific features influenced their work, they used many different access points from different producers. They observed that in close vicinity most of the bit errors were in the beginning of the packet, due to a backoff built in the protocols of WiFi. This lead them to the addition of multiple headers, which is possible without any additional implementation issues on the receiving side due to the design of 802.15.4. For 802.11b interference this led to an improvement of 30% while for 802.11g it led to 100% PRR with a preamble 9 Bytes. This lead to the decision to add a second header to the packet, which in turn added a problem with the CRC check, which has to be disabled. Therefore no hardware acks can be sent, as they rely on the hardware CRC check.

To deal with interference with the source being further away FEC was added, namely the Reed-Solomon codes. Their ratio of data to code was 65 to 30. Depending on how close the interferer is, 50 to 80% of the corrupted packets can be recovered. In their paper they omit to derive how many retransmission can be saved by the usage of BuzzBuzz, but rather use it in a test-setup using CTP together with BuzzBuzz. They say it reduced the amount of not acknowledged packets by 50% but do not make any comment on how many less retransmissions were needed, or on the exact test-setup, which is why we decided to look into that.

8.2 Constructive Interference

Using constructive interference in WSN was considered to be nigh impossible due to it needing tight time synchronization. Ferrari et al introduced Glossy as a protocol exploiting constructive interference and showing that achieving such synchronization is possible and provided for free with their protocol [Ferrari et al., 2011].

WSN radios use O-QPSK as modulation scheme to transmit its data. First the data is grouped into 4-bit symbols which are then mapped onto a pseudo-random noise sequence of 32 bits, so-called chips.

This sequence adds redundancy and WSN radios decide on a chip level rather than the signal itself. This fact enables radios to correctly detect chips up to a delay of $0.5\mu s$. This was first verified using simulations and the results, 98%, were beaten in actual experiments due to the capture effect.

In order to achieve time synchronization a 1-byte field is embedded into the message, called the *relay counter c*. The initiator sets it to zero while every subsequent node increases the counter. During a flood the packet length does not change, therefore the time needed to send the packet does not change either. This can be used to use c to calculate the time the flood was initiated, and also the time the packet has to be resent.

Glossy is radio driven and the network flooding needs to be decoupled from all other events, as it needs to be accurately periodic. This introduces the need for a scheduler, an operating system part not found in Contiki. The authors added their own scheduler and suggest the following architecture for the scheduler/operating system: ADDPIC. Interestingly this need for accurate periodic timings reminds of real-time operating systems. Contiki would surely benefit from having those capacities if needed. So far Contiki relies on cooperative scheduling, which in turn relies on the foresight of the developer to not block the CPU.

Glossy takes many measures to achieve its temporal accuracy, which is not easy as the clock drift of the Tmote Sky, which was used for the novel implementation, is pretty strong and temperature dependant. The authors managed to achieve good results for their flooding protocol: 99.9% packets were transmitted to all target nodes in tests without any interference. It would be really interesting how well Glossy does in harsh environments.

8.3 Handshakes under Interference

A general hard problem are handshakes or in other words: How to agree on a value using a noisy channel. [Boano, Zuniga, et al., 2012] investigated the possibility of jamming based agreements.

Most of the time it is not essential that the communication partner received the ACK to his message, but sometimes it is, which is then a hard problem, comparable to the *Two Generals Problem*, where two armies have to coordinate an attack. Therefore messengers, which can be caught on the way, are being sent to acknowledge a plan. If a messenger is caught, than the attack has to be aborted. In order to reach agreement an infinite number of acknowledge messages have to be sent figure 8.1. The probability to successfully transmit the last ACK can be increased by sending it k times.

Looking at WSNs or networks in general, n-way handshakes are used, which means that n messages are begin sent to ensure that both parties of the communication agree on the same value (TCP, WPA2). Generally a value is accepted if a party receives all of the intended messages, otherwise the value is discarded. This means, if the last message is lost the two parties disagree, under interference definitely possible.

In order to make sure both parties agree if a sender initiates an exchange the receiver answers with an acknowledge. In addition the sender, on reception of the acknowledge responses with a jamming signal. The receiver meanwhile observes the medium and concludes that if the medium has been busy for a certain time period, the sender must have received the acknowledgement. This works due to the fact that interference which is commonly found, such as WiFi, microwave ovens, Bluetooth, are all periodic and have short bursts of interference with pauses in between. If the medium is jammed longer than the longest common interference found in this environment it has to be part of the handshake.

While itself this handshake is not a protocol, it is intended to be used as a building block to enhance existing protocols, like FEC. Many protocols use acknowledgements in their design which would be a 2-way handshake. Many of those could benefit from JAG and should therefore be experimented with.

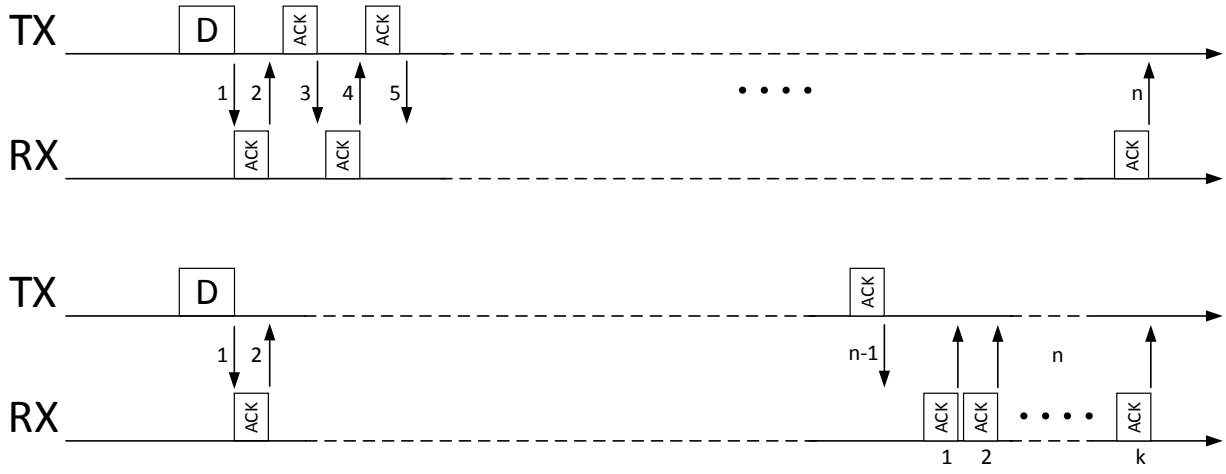

Figure 8.1: Two Generals Problem: The two generals use messengers which can be captured to communicate and try to agree on the time of a coordinated attack. The problem is similar to communication using a noisy channel. Each packet has a probability of failing, so both parties can only then be sure that a certain state is reached if an infinite number of ACKs is sent. The probability of the last ACK can be enhanced by sending it k-times.

8.4 Channel Hopping

While ContikiMAC has a good single-channel performance, it crumbles under heavy interference. MiCMAC was designed to take ContikiMAC's idea to multiple channels [Al Nahas et al., 2014]. Like ContikiMAC it uses two short consecutive CCAs to detect incoming frames. In addition to that every node switches channels (called channel hopping) using a pseudo random function with low computational cost.

Ideally a node always knows the channel and wake-up times of its communication partner, but if sending fails several times the sender needs to find the schedule of the receiver. The schedule is given by

$$X_{n+1} = (aX_n + c) \bmod N, \qquad n \geq 0 \tag{8.1}$$

where X is the channel, N the number of available channel, X_0 the seed, a and c a multiplier and the increment with $(0 \leq a, n < N)$ defining the channel hopping sequence. Importantly there is NO blacklisting built in the protocol, which means that every channel configured is being hopped to, regardless of its conditions. There can be short sequences (only using one $\langle c, a \rangle$) or long sequences using multiple configurations after another.

Similar to ContikiMAC, MiCMAC has a channel-lock phase in which communicating neighbours find out their schedule. The sender strobes one channel of the configured N channels and strobes it for one whole hopping sequence (see 8.2). After receiving the data-frame the receiver sends an acknowledgement-frame containing a, c and X_0. With this information the sender can compute the next channel and also has the wake-up time (phase lock) of its communication partner, therefore communications are know more efficient than before (see 8.3).

If the sender fails to receive an ACK after several tries on the expected channels of its communication partner it will start again with the initial rendezvous phase, otherwise it will always update its information on the receiver.

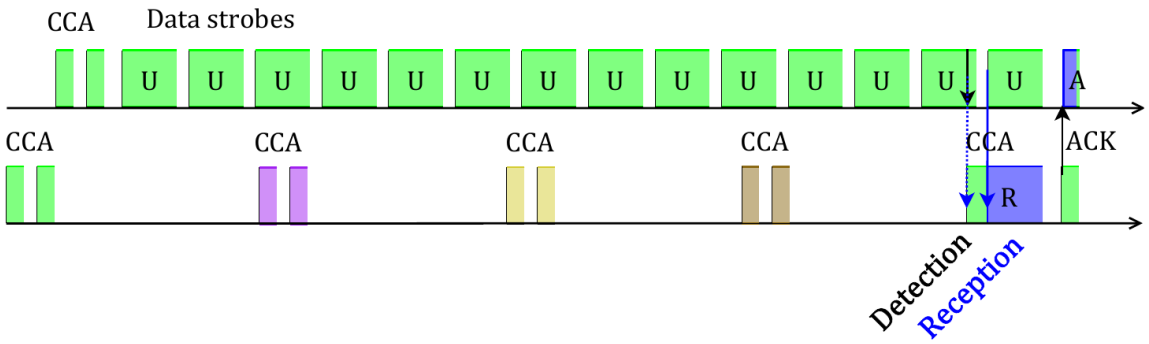

Figure 8.2: Initial rendezvous for MiCMAC with 4 channels. The sender repeatedly sends its data-frame on the same channel for a period of a whole pseudo-random sequence, a short one in this case. After successful transmission of the data-frame the receiver answers with an acknowledgement frame containing the channel configuration. Image extracted from Al Nahas et al. [2014, page 2]

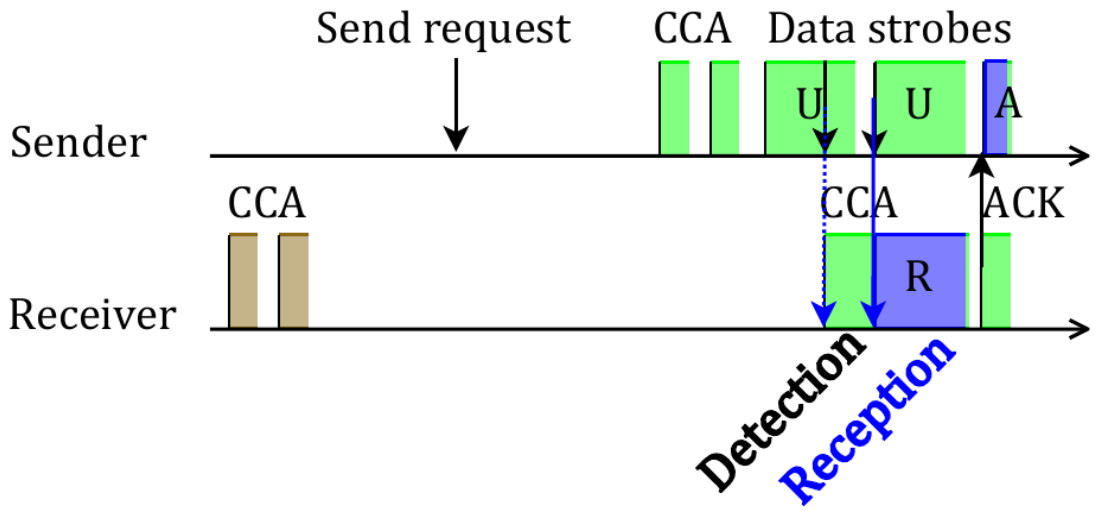

Figure 8.3: After reception of an ACK the sender can calculate the next channel and wake-up time of the receiver. Image extracted from Al Nahas et al. [2014, page 2]

8.5 Future Work

JamLab (see 3.1.1) was used for all of the experiments within this thesis and is in our opinion valuable for testbeds. Therefore it should be updated to newer hardware, such as the CC2650 used by the SensorTag. Maybe the addition of newer radios and faster CPUs will make it possible to emulate newer WiFi versions such as 802.11g/n which would be great for developing protocols resilient to interference.

Concerning ContikiMAC with FEC and JAG it should be possible to adjust the Phase Lock to work with the new and changed timings, this could make the protocol more energy-efficient.

As for the experiment framework, this could use a complete rewrite in e.g. Python with a database back-end to store all the experiment data. This should make it far easier to create plots, comparing different experiments and protocols with each other would then only be a matter of selecting the data in the database.

Bibliography

Advantic Sys [2015]. *CM 5000 MSP*. 2015. http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html (cited on page 8).

Al Nahas, Beshr et al. [2014]. "Low-power listening goes multi-channel". In: *Distributed Computing in Sensor Systems (DCOSS), 2014 IEEE International Conference on*. IEEE. 2014, pages 2–9 (cited on pages 57, 58).

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. Oct. 22, 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page vii).

Atzori, Luigi, Antonio Iera, and Giacomo Morabito [2010]. "The internet of things: A survey". *Computer networks* 54.15 (2010), pages 2787–2805 (cited on page 1).

Baccelli, Emmanuel et al. [2012]. *RIOT: One OS to Rule Them All in the IoT*. Research Report RR-8176. INRIA, Dec. 2012. https://hal.inria.fr/hal-00768685 (cited on page 10).

Beddit [2015]. *Beddit Sleep Tracker*. 2015. www.beddit.com (cited on page 2).

Boano, Carlo Alberto [2015a]. *Embedded Internet: Course Introduction*. Embedded Internet VU. 2015 (cited on page 7).

Boano, Carlo Alberto [2015b]. *Embedded Internet Lab*. 2015. https://online.tugraz.at/tug_online/lv.detail?clvnr=192785&cperson_nr=&sprache=2 (cited on page 8).

Boano, Carlo Alberto [2015c]. *Location Aware Computing: MAC Protocols*. Location Aware Computing LU. 2015 (cited on page 13).

Boano, Carlo Alberto [2016]. *Sensor Networks*. Sensor Networks VU. 2016 (cited on page 22).

Boano, Carlo Alberto, Thiemo Voigt, et al. [2011]. "Jamlab: Augmenting sensornet testbeds with realistic and controlled interference generation". In: *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE. 2011, pages 175–186 (cited on page 23).

Boano, Carlo Alberto, Marco Antonio Zuniga, et al. [2012]. "Jag: Reliable and predictable wireless agreement under external radio interference". In: *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*. IEEE. 2012, pages 315–326 (cited on page 56).

Buettner, Michael, Gary V. Yee, et al. [2006]. "X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: ACM, 2006, pages 307–320. ISBN 1595933433. doi:10.1145/1182807.1182838. http://doi.acm.org/10.1145/1182807.1182838 (cited on page 8).

Buettner, Michael, Gary V Yee, et al. [2006]. "X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM. 2006, pages 307–320 (cited on pages 16, 17).

Chinrungrueng, Jatuporn, Udomporn Sunantachaikul, and Satien Triamlumlerd [2007]. "Smart Parking: An Application of Optical Wireless Sensor Network". In: *SAINT Workshops*. IEEE Computer Society, 2007, page 66. ISBN 0769527574. http://doi.ieeecomputersociety.org/10.1109/SAINT-W.2007.98 (cited on page 1).

Deering, Stephen E [1998]. "Internet protocol, version 6 (IPv6) specification" (1998) (cited on page 13).

Dunkels, Adam [2011]. "The contikimac radio duty cycling protocol" (2011) (cited on pages 16–18).

Dunkels, Adam, Björn Grönvall, and Thiemo Voigt [2004]. "Contiki-a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pages 455–462 (cited on page 10).

Dunkels, Adam, Fredrik Österlind, and Zhitao He [2007]. "An Adaptive Communication Architecture for Wireless Sensor Networks". In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. SenSys '07. Sydney, Australia: ACM, 2007, pages 335–349. ISBN 9781595937636. doi:10.1145/1322263.1322295. http://doi.acm.org/10.1145/1322263.1322295 (cited on pages 12, 13).

Dunkels, Adam, Fredrik Osterlind, et al. [2007]. "Software-based On-line Energy Estimation for Sensor Nodes". In: *Proceedings of the 4th Workshop on Embedded Networked Sensors*. EmNets '07. Cork, Ireland: ACM, 2007, pages 28–32. ISBN 9781595936943. doi:10.1145/1278972.1278979. http://doi.acm.org/10.1145/1278972.1278979 (cited on page 12).

Durvy, Mathilde et al. [2008]. "Making Sensor Networks IPv6 Ready". In: *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. SenSys '08. Raleigh, NC, USA: ACM, 2008, pages 421–422. ISBN 9781595939906. doi:10.1145/1460412.1460483. http://doi.acm.org/10.1145/1460412.1460483 (cited on pages 8, 10).

Ferrari, Federico et al. [2011]. "Efficient network flooding and time synchronization with Glossy". In: *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE. 2011, pages 73–84 (cited on page 55).

GitHub [2015]. *Issuetracker Contiki, sorted by Author: Markus Schuss*. 2015. https://github.com/contiki-os/contiki/issues/created_by/schuschu (cited on page 9).

Gomez, Carles, Joaquim Oller, and Josep Paradells [2012]. "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology". *Sensors* 12.9 (2012), pages 11734–11753 (cited on page 15).

Guruswami, Venkatesan [2010]. *Introduction to Coding Theory: Reed-Solomon*. 2010. http://www.cs.cmu.edu/~venkatg/teaching/codingtheory/notes/notes6.pdf (cited on page 18).

Huang, Pei et al. [2013]. "The evolution of MAC protocols in wireless sensor networks: A survey". *Communications Surveys & Tutorials, IEEE* 15.1 (2013), pages 101–120 (cited on page 14).

Hurni, Philipp et al. [2011]. "On the accuracy of software-based energy estimation techniques". In: *Wireless Sensor Networks*. Springer, 2011, pages 49–64 (cited on page 12).

Le Dinh, Tuan et al. [2007]. "Design and deployment of a remote robust sensor network: Experiences from an outdoor water quality monitoring network". In: *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*. IEEE. 2007, pages 799–806 (cited on page 2).

Levis, Philip et al. [2004]. "The Emergence of Networking Abstractions and Techniques in TinyOS." In: *NSDI*. Volume 4. 2004, pages 1–1 (cited on page 10).

Liang, Chieh-Jan Mike et al. [2010]. "Surviving wi-fi interference in low power zigbee networks". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2010, pages 309–322 (cited on pages 45, 55).

Moteiv Corporation [2006]. *Tmote Sky Datasheet*. 1.0.2. 2006 (cited on page 8).

Müllner, Reinhard and Andreas Riener [2011]. "An energy efficient pedestrian aware Smart Street Lighting system". *International Journal of Pervasive Computing and Communications* 7.2 (2011), pages 147–161 (cited on page 1).

Ramesh, Maneesha V [2009]. "Real-time wireless sensor network for landslide detection". In: *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*. IEEE. 2009, pages 405–409 (cited on page 3).

Review, MIT Technology [2014]. *Smart Contraceptive with Remote Control*. 2014. http://www.technologyreview.com/news/528121/a-contraceptive-implant-with-remote-control/ (cited on page 2).

Richardson, Iain and Riley, Martyn [2016]. *Reed-Solomon Codes*. 2016. http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html (cited on page 18).

Roemer, Kay [2014]. *Location Aware Computing: MAC Protocols*. Location Aware Computing VU. 2014 (cited on page 14).

Texas Instruments [2015a]. *CC2650 SimpleLink Multistandard MCU*. October 2015. 2015. http://www.ti.com/lit/ds/symlink/cc2650.pdf (cited on page 8).

Texas Instruments [2015b]. *MSP430F1611*. 2015. http://www.ti.com/product/MSP430F1611 (cited on page 8).

Texas Instruments [2015c]. *SensorTag CC2650*. 2015. http://www.ti.com/diagrams/med_tidc-cc2650stk-sensortag_1_main.gif (cited on page 9).

Texas Instruments [2015d]. *SensorTag DevPacks*. 2015. http://www.ti.com/ww/en/wireless_connectivity/sensortag2015/devPacks.html (cited on page 9).

Tsiftes, Nicolas, Joakim Eriksson, and Adam Dunkels [2010]. "Low-power Wireless IPv6 Routing with ContikiRPL". In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '10. Stockholm, Sweden: ACM, 2010, pages 406–407. ISBN 9781605589886. doi:10.1145/1791212.1791277. http://doi.acm.org/10.1145/1791212.1791277 (cited on page 10).

Vasseur, Jean-Philippe and Adam Dunkels [2010]. *Interconnecting smart objects with ip: The next internet*. Morgan Kaufmann, 2010. Chapter 15, pages 199–230 (cited on page 13).

Zanella, Andrea et al. [2014]. "Internet of things for smart cities". *Internet of Things Journal, IEEE* 1.1 (2014), pages 22–32 (cited on page 1).