Tommy Sparber BSc

Experimental Analysis and Evaluation of RPL under Radio Interference

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Carlo Alberto Boano
TU Graz – Institute for Technical Informatics

Ass.Prof. Salil Kanhere
UNSW Sydney – School of Computer Science and Engineering

Graz, January 2017

AFFIDAVIT

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 used sources.
The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

....
Date Signature

Abstract

The routing protocol for low-power and lossy networks (RPL) is a routing protocol standardized by the Internet Engineering Task Force (IETF) to create a mesh network of wireless sensor nodes communicating via IPv6.
A large number of wireless sensor networks used to build IoT applications in the area of health care, smart cities, or smart production employs RPL to autonomously form a mesh topology and to route packets towards a gateway connected to the Internet.
Several of these applications impose strict requirements on network reliability and system lifetime. Achieving a long lifetime and a high reliability requires the system to be highly energy-efficient (in order to preserve battery capacity) and to be robust to external radio interference (in order to minimize packet loss and retransmissions).

Within this context, this thesis focuses on two main parts. First, an extensive experimental campaign investigates how RPL performs in the presence of radio interference. Experiments are performed on about 30 nodes in a wireless sensor network testbed using JamLab, a Contiki-based application that allows repeatable experiments with radio interference. The experiments demonstrate that RPL performs relatively well under interference as long as its underlying link-layer protocol (such as `ContikiMAC`) operates correctly. However, radio interference is shown to directly affect the operation of the clear channel assessment (CCA) mechanism, which may lead to a complete disruption of the network and to a high energy expenditure.

Building upon these results, the second part of this thesis proposes an improvement of `ContikiMAC` and shows, that by dynamically changing its CCA threshold, it is possible to improve network performance significantly even in the presence of interference. An experimental evaluation of the proposed approach shows that the packet reception rate of RPL with the improved `ContikiMAC` can be increased even in the presence of a malicious interference source and that the average energy consumption in the presence of a IEEE 802.11 device operating nearby can be reduced by up to 69 %.

Kurzfassung

Das Routingprotokoll für leistungsarme und verlustbehaftete Netzwerke (RPL) ist ein von der Internet Engineering Task Force (IETF) standardisiertes Routingprotokoll, um ein Mesh-Netzwerk aus kabellosen Sensorknoten, welche über IPv6 kommunizieren, zu erstellen.
Eine große Anzahl von kabellosen Sensornetzwerken, welche unter anderem zum Aufbau von IoT-Anwendungen im Bereich der Gesundheitsvorsoge, Smart Cities oder Smart Production errichtet werden, verwenden RPL, um autonom eine Mesh-Topologie zu bilden und Datenpakete in Richtung eines an das Internet angeschlossenen Gateways zu routen.
Einige dieser Anwendungen haben strikte Anforderungen an die Zuverlässigkeit und Lebensdauer des Netzwerks. Um eine lange Lebensdauer und hohe Zuverlässigkeit zu erreichen, muss das System sehr energieeffizient (um die Batterien zu schonen) und robust gegenüber externe Funktstörungen (um Paketverluste und wiederholte Übertragungen zu minimieren) sein.

In diesem Kontext ist diese Masterarbeit in zwei Teile gegliedert. Der erste Teil besteht aus einer umfangreichen experimentellen Untersuchung der Leistungsfähigkeit von RPL unter dem Einfluss von verschiedenen Arten von Funkstörungen. Die Experimente werden auf ungefähr 30 Sensorknoten in einer Sensornetzwerktestumgebung mit JamLab, einer auf Contiki basierenden Applikation, welche es ermöglicht, wiederholbare Experimente mit Funkstörungen durchzuführen.
Die Experimente zeigen, dass RPL auch bei Funkstörungen gut funktioniert, solange das darunter liegende Link-Layer-Protokoll (z.B. `ContikiMAC`) korrekt funktioniert. Es hat sich allerdings herausgestellt, dass Funkstörungen direkt den Clear-Channel-Assessment-Mechanismus (CCA) beeinflussen, was zu einer kompletten Unterbrechung des Netzwerks sowie zu unnötigem Energieverbrauch führt.

Aufbauend auf diesen Ergebnissen stellt der zweite Teil dieser Masterarbeit eine Verbesserung für `ContikiMAC` vor: Eine automatische Anpassung der CCA-Schwelle verbessert die Leistungsfähigkeit des Netzwerks signifikant, sofern Funkstörungen vorhanden sind. Eine experimentelle Evaluierung zeigt, dass die vorgeschlagene Methode die Paketempfangsrate von RPL mit `ContikiMAC` trotz starker bösartiger Funkstörungen erhöht und den durchschnittlichen Energieverbrauch, sofern sich ein IEEE 802.11 Gerät in der Nähe befindet, um bis zu 69 % verringern kann.

Acknowledgment

This master's thesis has been realized in the years 2015/2016 at the Institute for Technical Informatics at the University of Technology in Graz, Austria as well as at the School of Computer Science and Engineering at the University of New South Wales in Sydney, Australia.

First of all I would like to thank my supervisor Prof. Carlo Alberto Boano for always having the patience to support me and my thesis. Without his ongoing feedback and reviews, I would have never been able to write and form the present thesis as it is now.

A very big thank you is also directed towards Prof. Kay Römer for connecting me with Prof. Salil Kanhere from the UNSW in Sydney, Australia. This effort made it possible to study abroad in Sydney for six months together with my wife. We collected many great impressions and experiences while visiting this beautiful city and country.
Thank you Salil, as well as Carlo for supporting us with all the formalities and paperwork that have been required. These six months are a never forgettable experience.

Lastly, I would like to thank my wife, friends and family for supporting me throughout all the years studying in Graz.

Graz, January 2017 Tommy Sparber

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

μC microcontroller 19–23, 25

6LoWPAN IPv6 over Low power Wireless Personal Area Network 7, 14, 27, 31, 33, 34, 44

ACK acknowledgment 36, 45, 63, 93

ADC analog-to-digital converter 20

AEDP Adaptive Energy Detection Protocol 90, 92–94

AP access point 39, 91

API application programming interface 59, 78

AutoCCA automatic adaptive CCA threshold adjustment algorithm 8, 9, 75, 77, 82–85, 87–89, 94

BSL bootstrap loader 20, 23

CA collision avoidance 92

CCA clear channel assessment 2, 3, 8–11, 16, 17, 36, 37, 43, 47, 50, 61–63, 65–67, 71, 73–75, 77, 78, 81–83, 85–88, 90, 92–94

CCR Channel Check Rate 7, 37, 46, 49, 55, 61

CLI command-line interface 24, 25

CPU Central Processing Unit 14, 19, 21, 29, 44, 49, 56, 65, 94

CSMA carrier sense multiple access 31, 36

CSV comma-separated values 58

CTP Collection Tree Protocol 90, 91, 93

DAG directed acyclic graph 35

DAO destination advertisment object 35

DIO DODAG information object 35

DIS DODAG information solicitation 35

DODAG destination oriented directed acyclic graph 7, 35, 45

ECG electrocardiography 15

ETX expected transmission count 8, 9, 36, 46, 47, 56, 61–65, 67–70, 82, 91–93

GCC GNU Compiler Collection 20

GPIO general-purpose input/output 22

HTML HyperText Markup Language 7, 56

HTTP Hypertext Transfer Protocol 59

I^2C Inter-Integrated Circuit 20

ICMP Internet Control Message Protocol 8, 27, 33, 35, 44, 71

IEEE 802.11 Wi-Fi 2, 3, 16, 17, 91, 93, 95

IETF Internet Engineering Task Force 2, 3, 14, 34, 35, 94

IoT Internet of Things 2, 3, 14

IP Internet Protocol 33

IPv4 Internet Protocol Version 4 33

IPv6 Internet Protocol Version 6 2, 3, 7, 14, 31–34, 41, 94

IrDA Infrared Data Association 19

LQI link quality indicator 22, 92, 95

MAC Media Access Control 21, 31, 34, 36, 57, 75, 91, 92

MTU maximum transmission unit 34

OF objective function 9, 35, 36, 43, 46, 61, 63, 65, 67, 82, 90–92, 95

OF0 objective function 0 8, 9, 36, 46, 47, 61–65, 67, 69, 70, 82, 91, 92

OS operating system 26, 27

PC personal computer 25, 26, 33

PCB printed circuit board 21, 25

PR pull request 17, 41

PRR packet reception ratio 7, 8, 15, 17, 44, 46–48, 56, 61–63, 65–67, 70, 71, 74, 77, 81–83, 85, 87–94

QoS quality of service 33

RAM Random Access Memory 19, 20

RDC radio duty-cycling 8, 17, 31, 33, 36, 45, 46, 49, 61–63, 65, 78, 92

RISC reduced instruction set computing 20

ROLL Routing Over Low-power and Lossy networks 35

RPL routing protocol for low-power and lossy networks 2, 3, 7–10, 14, 16–19, 23, 25, 27, 31–33, 35, 41–47, 56, 61–65, 70, 71, 75, 85, 87, 88, 90–95

RSSI received signal strength indicator 7, 8, 16, 22, 37, 39–42, 45, 47, 48, 50, 51, 57, 62, 63, 66, 73–77, 79, 80, 88, 93, 95

SMA SubMiniature version A 21, 25

SNR signal-to-noise ratio 15

SPI Serial Peripheral Interface 20, 22, 23

SSH Secure Shell 25

TCP Transmission Control Protocol 27, 33

TI Texas Instruments 20, 21, 27

TXP transmission power 7, 47, 48, 51, 52, 61, 91

UART universal asynchronous receiver/transmitter 20, 23, 26

UDP User Datagram Protocol 27, 32, 33, 44–46

USB Universal Serial Bus 23, 25, 26

VoIP Voice over IP 33

WSN Wireless Sensor Network 14–16, 18–21, 23, 27, 35, 38, 47, 75, 90, 92, 94

Chapter 1

Introduction

The Internet of Things (IoT) is not only the future, but it is already a reality. The idea behind the IoT is to connect small embedded devices (also called “things") to the Internet without requiring user interaction. Billions of “things" are already connected to the Internet and interact with each other and their surrounding environment: their number should exceed 20 billion by 2020 according to Gartner[1].
A big part of the IoT are Wireless Sensor Networks (WSNs): networks of small battery powered devices, also called nodes, equipped with a low power Central Processing Unit (CPU), a radio to communicate over a wireless link, and some type of sensor or actuator. The measured data often needs to be collected at a central location. In the IoT context, one option is to allow direct connection of these nodes to the Internet using the Internet Protocol Version 6 (IPv6). To facilitate this connection, the Internet Engineering Task Force (IETF) standardized 6LoWPAN as a protocol to implement IPv6 in a low power wireless personal area network such as a WSN by performing header compression and packet fragmentation.
As WSN are typically using mesh topologies, where data is passed over multiple hops to a central node (sink), a routing protocol needs to be employed to find a suitable path (route) from each node to the sink. The IETF recently standardized the routing protocol for low-power and lossy networks (RPL). This protocol is especially designed for WSNs and will be used in this thesis to evaluate and improve its performance while the network is subjected to radio interference.
RPL is used to automatically form a network and can be used in both dense and sparse WSN deployments.
In the past decade, a large number of WSN applications have been deployed, ranging from agriculture, health care and industry to smart cities, smart cars and smart grid.

- *Agriculture*
  A team of researches in Australia built a WSN to monitor the movement and spread of the cane toad [18] (i.e., the so called cane toad monitoring). This toad was introduced to Australia in the 1930s to control pests in the sugar cane crops, but, as it has no natural predators, it grew to a pest itself. The WSN in this application employs acoustic sensors to sample the croaking of the toads. These audio samples are then

[1] http://www.gartner.com/newsroom/id/3165317

analysed and classified to distinguish between different frog species. This data is then used to estimate the cane toad population and movements.

- *Health Care*
  In health care, the development of embedded systems measuring vital signs such as pulse monitors, blood pressure, blood glucose and electrocardiography (ECG) allows to monitor the health conditions of people in needs. A WSN worn inside (implanted) or on the body enables the patient to move freely while the data is collected using wireless transmissions. Reporting the data to an Internet-connected server that processes the data allows for an early disease detection and prevention [27].

- *Smart cities*
  A typical application in a smart city is the control of street lighting intensity through the use of a WSN in order to reduce the energy consumption. Each lamp acts as a node, but only some nodes also employ a Doppler sensor to detect road traffic. Upon a traffic detection, the information is passed to the other lamps and the light intensity is increased [21].
  In an smart parking application, each parking lot is equipped with a sensor node that detects if the lot is free or occupied. This data is then transmitted using the wireless link to a central Web-server which allows the vehicle driver to find a vacant parking lot using a smart phone [35].

- *Smart home*
  WSN play an integral role also in smart home applications. Various devices like lamps, heating, or blinds can work together with sensors for temperature and utility meter readings to improve the everyday life of a resident.

It is important to highlight that applications in health care and smart cities are safety-critical, i.e., a failure may endanger lives and hence the underlying WSN communication needs to be reliable. Besides safety-critical applications, a reliable system is also important for its acceptance and success. An unreliable system, for example an unreliable smart home application, would lower the user satisfaction and thus limits its success.
The reliability of a WSN strongly depends on its ability to successfully communicate. Changes of the environment surrounding the deployed network have a big influence on its capability to communicate. High temperature changes have an impact on the stability of the clocks used in these embedded devices and on the amplifiers in the radio module [4]. This reduces the ability to synchronize nodes and also the transmitted and received signal strengths, which then reduces the nodes capacity to successfully receive a packet.
Beside temperature, a main obstacle to reliability is **radio interference**. The latter lowers the signal-to-noise ratio (SNR) as the noise level is increased. When the SNR approaches the zero, the radio module looses its ability to distinguish between a valid signal and noise, leading to packet loss and thus to a decrease in packet reception ratio (PRR). Besides lowering the PRR, also the latency and energy consumption increase, as packets have to be retransmitted and the radio has to be kept on for a longer time. A reliable network is characterized by a high PRR, low latency, and a low energy consumption even when subject or exposed to radio interference.

WSNs should be reliable both under common radio interference from surrounding wireless devices and from intentional "malicious" attacks. Common radio interference is a problem that derives from the fact that the frequency spectrum is shared between multiple users. Many devices and protocols such as Wi-Fi (IEEE 802.11), Bluetooth, ZigBee, and home applications such as microwave ovens use frequencies in the license free 2.4 GHz band. IEEE 802.11 devices usually have a higher transmission power than the low power radios used by WSNs. The number of devices using the same band is increasing: IEEE 802.11 is widespread in every household: in the earlier days a family had one device, but nowadays each member has its own notebook, tablet and/or smart phone. The increased usage of the 2.4 GHz band, increases the probability for a WSN to be subject to radio interference. Indeed, beside common radio interference, there might also be a intentional malicious attacks that try to sabotage the network by generating artificial noise or by flooding it with useless packets [25].

1.1 Problem statement

As radio interference has a huge influence on the ability to communicate with other nodes, it is important to understand how RPL and the other communication layers are affected by radio interference and whether something can be done to improve the situation. Depending on the strength of interference, the node may not be able to communicate at all, or may exhibit a higher energy consumption and latency due to necessary retransmissions.
An important part of the nodes communication is the clear channel assessment (CCA). The CCA is used to ensure that no-one else is sending a packet at the same time, i.e., to verify that the radio channel is clear (for collision avoidance), and to check if there is something to be received (when looking for incoming packets). Hence radio interference may affect the CCA both before sending and while receiving a packet. Before sending, the node sees the radio channel as always busy and no packets are sent. While receiving, the radio is kept on for a longer period as there might be a packet transmitted and thus the node's battery drains faster.
In typical configuration, the CCA checks if the received signal strength (RSSI) is above a *fixed* threshold at which the channel is considered busy. If the signal strength of the neighbouring nodes is above the interference noise level they would be able to communicate nevertheless, but, as the channel is considered busy, no packets are sent.
In this thesis we want to show that, by adjusting the CCA threshold dynamically between the noise level and the RSSI of the packets from the neighbouring nodes, it is possible to improve the ability to communicate and thus the overall network reliability without any significant tradeoffs.

1.2 Contributions

This thesis investigates how RPL performs in the presence of radio interference experimentally on real hardware in a testbed with around 30 nodes. The interference is generated using JamLab, a Contiki OS application that uses the test mode of the radio transceiver to create artificial, but repeatable interference. JamLab is used to show that RPL performs well under interference as long as the underlying protocol (such as `ContikiMAC`) performs

well.
The large experimental campaign shows indeed many lessons. The most important one being that the used link-layer `ContikiMAC` with its fixed CCA threshold is easily blocked by radio interference. To fix this problem, this thesis proposes to dynamically change the CCA threshold of the radio duty-cycling (RDC) in order to achieve significant improvements on the reliability of communications.
The energy consumption under interference is reduced up to 65 % under heavy interference (i.e., in the presence of a IEEE 802.11 device performing file transfer), while the PRR is increased by up to 60 % in the presence of a continuous carrier jammer (i.e., in the presence of a deliberate attacker).
In addition to these fundamental contributions, small improvements to the Contiki OS have been made available as pull requests (PRs)[2].

Methodology. All experiments in this thesis have been carried out on real-world testbeds as part of a large experimental campaign. In particular, more than 200 individual experiments have been run. The log files and generated plots sum up to more than 21,700 files for a size of 13.9 GB. 90 different run configuration scripts were created and are accompanied by 47 compiled firmware files. The tooling created consists of more than 55 Python classes and shell scripts (3400 lines of code) to manage the experiment runs, parse the collected data and generate the plots, tables and statistics used in this thesis in a semi-automatic fashion.
According to its logfile, the testbed has been reserved for this work about 140 times for a total duration of more than 1500 h (although the testbed has not been continuously used during a reservation).

1.3 Structure

This thesis is structured as follows. Chapter 2 describes the relevant background to understand the technical contribution. In Chapter 3 follows the experimental campaign showing how to approach the problem of investigating RPL under radio interference using multiple experiments on a real testbed. This section also contains the results and lessons learned from the extensive evaluation. Chapter 4 describes the design and implementation of the adaptive CCA threshold algorithm, implemented to solve the problems highlighted in Chapter 3. An evaluation of the implemented adaptive CCA algorithm is presented in Chapter 5. Related work in the area is described and summarized in Chapter 6. This thesis concludes with Chapter 7, which recaps the contributions and the results and gives an outlook on future work.

[2] https://github.com/contiki-os/contiki/pulls?q=is:pr+author:tsparber

Chapter 2

Background

This chapter provides an overview and background information of the technologies used to test RPL subjected to radio interference, to identify the problems observed and finally, propose an improvement.

2.1 Wireless Sensor Network

A Wireless Sensor Network (WSN) is a collection of small embedded devices, called nodes, deployed with the goal to collect some sensor data from their physical environment. The measured data is then sent to a base station using a wireless connection. Typically the base station is not directly reachable and the data must be retransmitted from one node to the next one. An example network with multiple hops, also called “mesh" network, is shown in Figure 2.1.
A long lifetime of each node is expected. As these devices are usually battery powered, special care to the energy consumption has to be taken.

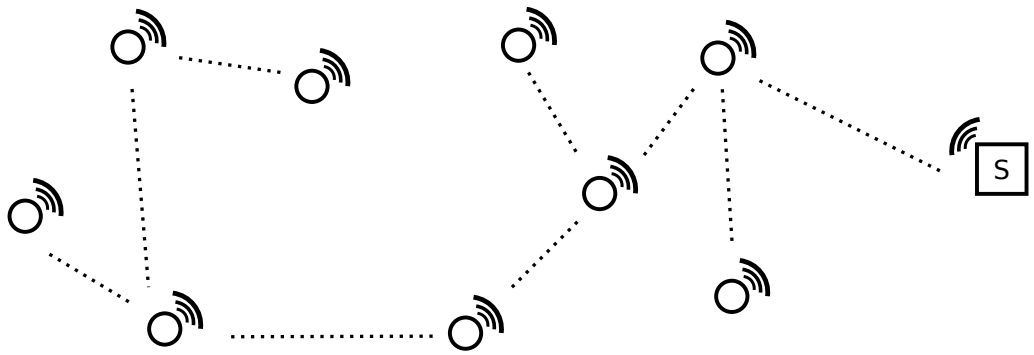


Figure 2.1: A typical architecture of nine nodes with one sink which collects the data.

Important and typical properties of these devices are:

- *Cheap*
  The devices should be cheap, so that a large number of them can be deployed to function as a macroscope. Whereas a microscope allows to observe the tiniest object, a macroscope enables researchers to study large scale phenomena. Some examples

are: Monitoring the vibrations of a bridge [34], tracking the spread of the cane toad in Australia [18], or learning about the movements of glacier snouts [14].

- *Small*
  Each node has typically a volume of less than 1 dm^3, batteries included. In the earlier days devices as small as dust, so called “smart dust" were envisioned [20]. These specks of dust could then be deployed by air or mixed into colours used to paint the surface of interest. Devices the size of dust exists only as prototypes, are very hard to produce, and are severely limited in terms of communication possibilities and power. In general, WSNs focus on larger physical sizes, built with readily available microprocessors.

- *Battery powered*
  A long lifetime in the range of a couple of months to multiple years is expected. Changing batteries in hundreds or thousands of nodes deployed is quite expensive and time consuming. To achieve a long battery life, a low energy consumption needs to be considered in every aspect of the design and protocols used. RPL is especially designed for networks where the nodes are sleeping most of the time and only wakeup periodically to receive and transmit packets. The sleep and wakeup cycle needs to be adjusted to fit the application needs, as low energy consumption and high responsiveness are contradicting requirements.

- *Low power wireless radio*
  Using low power wireless communication is a key feature of a WSN. Typically this communication happens using radio waves. Using optical links such as Infrared Data Association (IrDA), would be possible but requires a line of sight, which is often not given.
  To simplify the development and deployment of WSNs, license free ISM frequency bands are used. Many nodes communicate in the 2.4 GHz band which is usable world-wide, but communication is also possible using other frequencies such as 868 MHz in Europe and 915 MHz in the USA.
  A common standard among WSNs is to use a IEEE 802.15.4 compatible radio, this standard is also known as the base for “ZigBee". This thesis also uses a IEEE 802.15.4 radio, but not the upper layers defined by ZigBee.
  A radio in a WSN uses a low transmission power and also the algorithms used are less sophisticated then those used for example in a Wi-Fi network. An algorithm optimized for high data throughput while being robust to radio interference requires significant more energy to process the received signal. Even in a low power IEEE 802.15.4 radio the energy required while listening and receiving a packet might be higher than those needed to transmit a packet (depending on the transmission power).

- *Low power CPU*
  The nodes in a WSN typically use a low power microcontroller (μC). They might range from an 8-bit AVR to an 16-bit MSP430 as used in this thesis or to the latest devices including an 32-bit ARM microprocessor. The CPU has a limited amount of Random Access Memory (RAM), usually only some kilobytes and less than 1 MB of flash for the firmware.

- *Sensors*
  Nodes in a WSN are typically used to measure and report something of their physical environment. This includes, but is not limited to the connection of sensors to measure temperature, humidity, pressure, light, speed, distance, vibration, orientation, magnetism, audio and video.

- *Redundancy*
  A node might fail due to various reasons, such as production errors, software failures, running out of battery or physical damages due to environmental impact. For example, the case might not be perfectly water proof which leads to short circuits and corrosion [2]; a faulty battery might leak due to high temperature impact; vibrations might force soldered components to dislocate.
  Therefore, deploying some redundant nodes, at least to have multiple options while routing data is important. The routing protocol is then able to select from multiple nodes and paths to reach the sink even if one node fails or is not able to communicate due to interference.

2.2 Hardware

All the implementation and evaluation presented in this thesis is performed on real hardware in a testbed located in Graz, Austria. The sensor node and testbed used are described next.

2.2.1 TelosB / Tmote Sky

This thesis uses MTM-CM5000-MSP sensor nodes which are manufactured by Maxfor and sold by Advanticsys[1]. This node is based on the famous open source TelosB / Tmote Sky designed by University of California, Berkeley.
The Tmote Sky features a Texas Instruments (TI) MSP430 μC, a CC2420 radio, integrated sensors to measure temperature, humidity and light. Further it has a built in FTDI USB-To-Serial converter that allows a direct connection to a PC for debugging and firmware upgrades. Figure 2.2 shows the used MTM-CM5000-MSP replica of a Tmote Sky node. The typical operating conditions are shown in Table 2.1.
A simplified block diagram of the mote is shown in Figure 2.3. The most important components are:

- *CPU: TI MSP430 F1611*
  The TI MSP430 F1611 is a 16-bit RISC mixed signal microcontroller [30]. It has 48 kB of program flash memory and 10 kB of RAM. The operating frequency is up to 8 MHz. Integrated peripherals include an analog-to-digital converter (ADC), two Serial Peripheral Interfaces (SPIs), Inter-Integrated Circuit (I^2C), two universal asynchronous receiver/transmitters (UARTs) and two 16-bit timers.
  To reprogram the μC in system a bootstrap loader (BSL) is included, which allows writing of a new user application using the UART interface. A free and open source toolchain based on the GNU Compiler Collection (GCC) is available.

[1] http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html

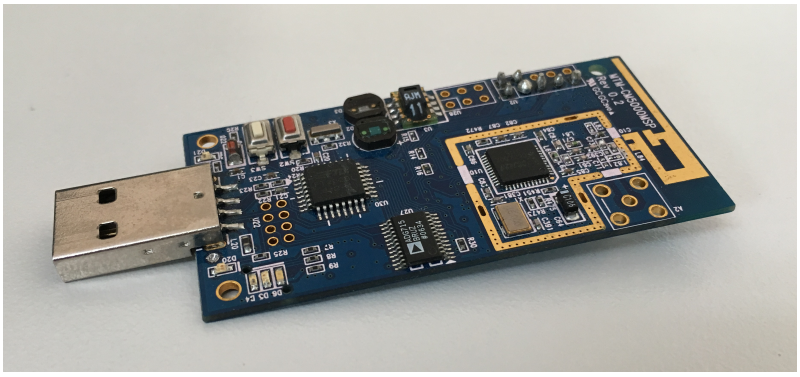


Figure 2.2: MTM-CM5000-MS, a TelosB / Tmote Sky replica.

	MIN	NOM	MAX	UNIT
Supply voltage	2.1		3.6	V
Supply voltage during flash programming	2.7		3.6	V
Operating free air temperature	−40		85	°C
Current Consumption: μC on, Radio RX		21.8	23	mA
Current Consumption: μC on, Radio TX		19.5	21	mA
Current Consumption: μC on, Radio off		1800	2400	µA
Current Consumption: μC idle, Radio off		54.4	1200	µA
Current Consumption: μC standby		5.1	21.0	µA

Table 2.1: Typical Operating Conditions (Source: [24]).

Five power-saving modes allow the ultra low power CPU to reduce its energy consumption depending on required functionality. An interesting feature for WSN is the possibility to wake-up from standby mode in less than 6 µs.

- *Radio: TI CC2420*
  The Tmote Sky's radio is a TI CC2420 2.4 GHz IEEE 802.15.4 ZigBee-ready RF transceiver [31]. The radio handles the physical and some parts of the Media Access Control (MAC) layers of the IEEE 802.15.4 standard. The maximum achievable data rate is 250 kbps.
  The radio operates in the 2.4 GHz band and supports the 16 physical channels defined by IEEE 802.15.4. Each channel has a bandwidth of 3 MHz. They are numbered from 11 to 26. Figure 2.4 shows how the IEEE 802.15.4 / ZigBee channels align with the Wi-Fi channels.

  All the experiments in this thesis will use channel 26, so that the overlap with normal Wi-Fi and the radio interference is minimized.
  The CC2420 has the possibility to select its transmission power from 32 different levels. The current consumption while transmitting depends on the selected transmission power. Table 2.2 lists the different power levels, its expected output power and current consumption while transmitting. The Tmote Sky includes a PCB Antenna and an optional SMA connector for an external antenna.

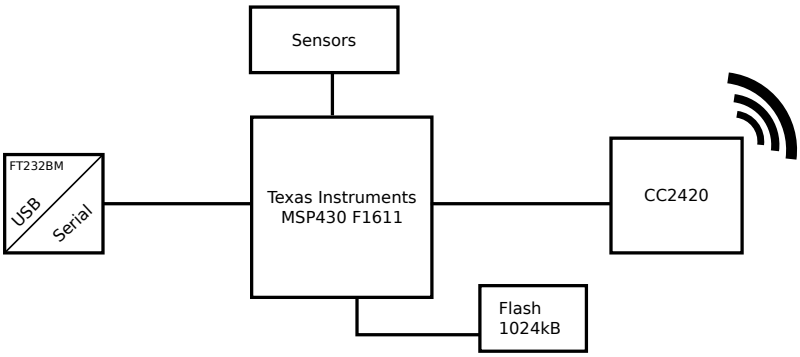


Figure 2.3: A simplified block diagram of a Tmote Sky.

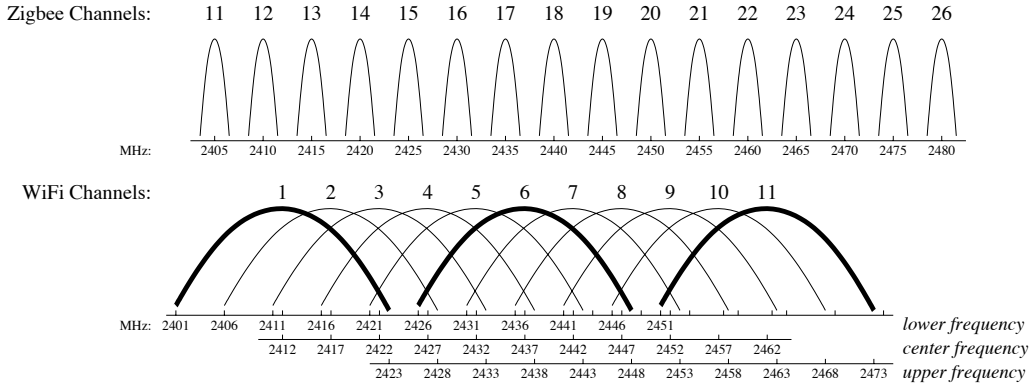


Figure 2.4: IEEE 802.15.4 / ZigBee channels compared to Wi-Fi channels in the 2.4 GHz band. (Source: [26]).

The connection between the μC and the CC2420 is established using SPI and some general-purpose input/outputs (GPIOs).
To estimate the link quality, the CC2420 includes two hardware indicators. The link quality indicator (LQI) gives a numeric value between 60 and 110 where 60 means a poor link and 110 a good one. The LQI is not used in this thesis. The other indicator is the received signal strength indicator (RSSI). Here the energy of the packet received is measured and reported as number between 128 and -128 dBm, but typically a value between 0 and -95 dBm is expected. A higher RSSI indicates a stronger signal and thus a better link. The RSSI can also be measured any time to determine the current noise floor or that currently a packet from another node is transmitted.

- *Sensors: Temperature, Light (2x), Humidity, Buttons*
  The Tmote Sky includes a combined temperature and humidity sensor (SHT11) built by Sensirion, as well as two light sensors built by Hamamatsu measuring the total solar radiation (S1087-01) and the other one only the photosynthetic active radiation (S1087) respectively. Beside the sensors, there is also one user programmable button and a reset button.
  This thesis does not use any of the sensors, as the focus is on the communication

PA_LEVEL	**Output Power [**dBm**]**	**Current Consumption [**mA**]**
31	0	17.4
27	−1	16.5
23	−3	15.2
19	−5	13.9
15	−7	12.5
11	−10	11.2
7	−15	9.9
3	−25	8.5

Table 2.2: Output power settings and typical current consumption of the CC2420 radio (Source: [31]).

rather than measuring something specific.

- *External Flash*
  The Tmote Sky nodes also include an external 1024 kB Flash, the ST M25P80. It is connected over SPI to the μC and can be used to store for example measurement data. The flash is not used in this thesis.

- *USB-Serial Converter*
  To convert the UART signals to Universal Serial Bus (USB), a FTDI FT232BM chip is mounted. This IC provides a virtual COM device to the connected PC. The USB connector is also used to power the board when no batteries are present.
  The USB connection can be used to either program a new firmware using the BSL or to read and write debug messages from nodes the application code.

2.2.2 Experimentation on Testbeds

This thesis studies RPL on real hardware, to test on a large scale, using a testbed is a good idea. There are several locations which can be used to test on a larger scale. The public testbeds typically have an online reservation and test management system. The terms and conditions of each testbed are different, but in general they are free to use for research purposes.
In a testbed, the nodes are connected either directly, or via special hardware to a central server using a back channel such as the USB connection of the Tmote Sky. The server allows for new firmware distribution and collection of log files. This allows for faster program-compile-run-cycles compared to when each node would be programmed separately.
Following next is a list of testbeds that were considered for this thesis, as they are all based on Tmote Sky or similar nodes. Some testbeds are not available any more, but were referenced in other publications on WSN. A more comprehensive survey on testbeds can be found under [17].

- *MoteLab*
  MoteLab was one of the first public available testbeds. It was built at the Harvard University and first published in 2005 [33]. It consisted of about 190 Tmote Sky sensor nodes. The testbed is currently not available, the future status is unknown.

The last time the homepage was reachable, according to the Internet Archive was in 2012[2].

- *NetEye*
  NetEye was a testbed at the Wayne State University that featured 130 Tmote Sky nodes [28]. It was published in 2008, but registration[3] is not possible any more. It is also officially considered unavailable.

- *Kensei / KanseiGenie*
  The Kansei testbed provided 384 Tmote Sky nodes, but is also not connected anymore[4]. It was first published in 2006 [1] and built at the Ohio State University.

- *Wisebed*
  The Wisebed was a project to connect multiple testbeds using a federated web API [7] that ended in 2011. The project connected nine testbed which could then be accessed using the same interface[5]. When starting with this thesis, the testbed located at the University of Lübeck (UZL) still provided access to around 54 Tmote Sky nodes.
  The reservation and test management could be made either using a Web interface or automated using a command-line interface (CLI). Also the output of each node could be read in real time or collected after the test had finished. Only some initial tests learning the interface could be performed, then the testbed had technical problems and is now not available anymore.

- *Indriya*
  This testbed located in Singapore provides access to about 100 Tmote Sky nodes across three floors [8]. The reservation and scheduling of tests is controlled using a Web interface[6]. Access to the log files is possible using a direct connection to a MySQL database. The tests are limited to 30 min per day. As this is very limiting, this testbed was not used as part of this thesis.

- *TWIST*
  The TWIST testbed is developed at the Technische Universität Berlin and provides access to 102 Tmote Sky nodes [16]. The Web interface[7] allows to book the testbed for a maximum of 48 hours per week. The configuration is then performed either online or through the use of a CLI. They are focused on running TinyOS applications, but running a Contiki application is also be possible, but requires changes to the serial output, as the TinyOS SerialForwarder is used by default.

- *FIT/IoT-LAB / SenseLAB*
  The FIT IoT-Lab[8], renamed from SenseLAB [9], is a large-scale Internet of Things testbed which connects six individual testbeds to one large federated tested featuring about 2700 nodes. All the testbed sites are located in France. Experiments are either

[2] https://web.archive.org/web/20120106040115/http://motelab.eecs.harvard.edu/index.php
[3] http://neteye.cs.wayne.edu
[4] http://kansei.cse.ohio-state.edu/KanseiGenie
[5] http://wisebed.eu
[6] https://indriya.comp.nus.edu.sg
[7] https://www.twist.tu-berlin.de
[8] https://www.iot-lab.info

managed using a Web interface or CLI tools. The testbed allows measuring the voltage and current consumption of each node during an experiment. They feature different processor architectures, but no Tmote Sky nodes. Their WSN430 node uses the same μC and radio as a Tmote Sky and a port for Contiki exists, but to use a single platform for consistency, this testbed will not be used.

- *FlockLab*
  The FlockLab is located at the ETH Zürich [22]. It provides specialised nodes, called observers, which allows tracing of signals and measuring the power consumption in real time. The observer hardware allows connection of up to 4 different nodes using custom made target adapter. Currently, each observer has at least one Tmote Sky attached, so tests on Tmote Sky nodes can be performed. Most nodes are located indoors, but some of them are also outdoors. The testbed has about 30 observer which can be reserved using a web interface.[9] The test management happens through the creation of a XML file which is then uploaded. There are also some CLI tools provided, which allow automated reservation and provisioning of the nodes. The output of the nodes debug interface is collected across all observers and stored in a single file.

- *TempLab*
  The TempLab is located at the University of Technology in Graz [5]. It is not available for general public use, but access to it can be requested. The testbed consist of two rooms, called LAB North and LAB South. The LAB North has 60 Tmote Sky nodes, numbered from 100 to 159. Whereas the nodes from 136 to 159 are position close together in one corner, the other nodes are spread across the room according to the map shown in Figure 2.5. Each of the nodes in the LAB North uses the PCB antenna to communicate. The LAB South has 17 Tmote Sky nodes distributed in a similar fashion to the LAB North, but they use an external antenna connected to the SMA connector. This increases the received signal strength of received packets and, as the nodes are quite close to each other, even with a low transmission power all the nodes are connected directly.
  This thesis uses only the LAB North with the PCB antenna because to test RPL properly, a multi hop network needs to be formed where not all nodes are directly connected.
  To access the testbed, it is required to establish a Secure Shell (SSH) connection to the personal computer (PC) located in the lab. Every node is connected to this PC via USB. This connection is established either directly or by means of USB-Hubs and Ethernet-to-USB extender cables.
  The testbed provides three scripts that are used to control an experiment:

  - `experiment_start.sh`
    This script starts a new experiment: first it checks if the testbed is already used, and, if not, the testbed is reserved for the current user.
  - `experiment_stop.sh`
    This script is used to stop a reservation. If the testbed is used by the current user, it is then released and available again for other users.

[9] https://www.flocklab.ethz.ch

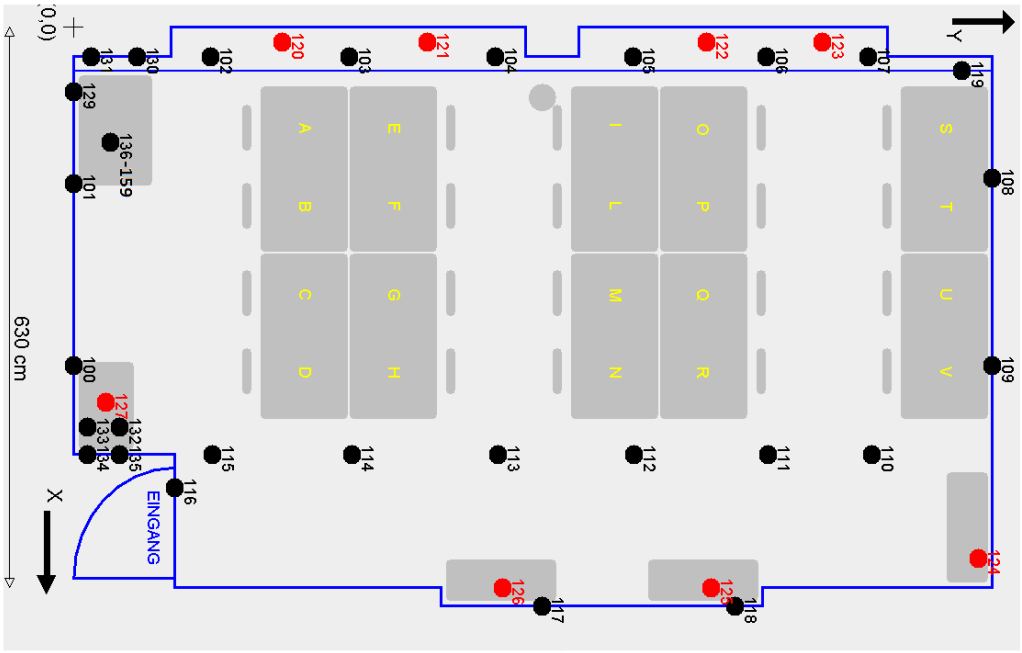


Figure 2.5: Map of the TempLab LAB North testbed.

– `experiment_list.sh`
  This script lists the current user of the testbed and also the history of the last 10 reservations.

As each node is connected using USB, the FTDI USB-Serial-Converter allows direct access of the UART interface of the Tmote Sky node. The same tools as if the nodes would be connected directly to the developers PC can hence be used (see Section 2.3 for further details). To simplify the access, each Tmote Sky, which is identified by its serial number by the operating system (OS), is mapped using a udev rule to a device under `/dev/mote*` to its corresponding node id. The nodes 100 to 159 from the LAB North are mapped to `/dev/moteN0` ... `/dev/moteN59`, the nodes in the LAB South from node ID 200 to 216 are mapped to `/dev/moteS0` ... `/dev/moteS16`.
To control collecting data from the nodes, the testbed provides another three scripts:

– `traces_start_tty.sh`
  This script starts a new collection of the nodes debug output. For each node, a separate `serialdump` process is created that logs the received UART data to one text file.

– `traces_close_tty.sh`
  This script is used to close a trace: every `serialdump` process is killed and the text files are moved to a new directory under the `traces` folder.

– `traces_remove_tty.sh`
  To discard the traces this script can be used: in a similar fashion as the previous script, this script kills all the `serialdump` processes, but then deletes all the recorded text files.

2.3 Contiki OS

As an operating system for Wireless Sensor Network the open source Contiki OS[10] was chosen. It has really good support for the Tmote Sky nodes and is also widely used for networked embedded system. The OS is written in C and includes everything needed to create a network of nodes build on RPL and 6LoWPAN.
The project was started in 2002 by Adam Dunkels and is still under active development[11]. Contributions to the code base are made by a high number of individual developers and also some companies such as TI and STMicroelectronics.
It is designed to run on a multitude of platforms and processor architectures. The supported processor architectures range from ARM, AVR, x86 to more exotic ones such as 6502 and Z80, of course also the MSP430 is supported as used in the Tmote Sky nodes.
The lightweight uIPv6 Stack supports Internet Control Message Protocol (ICMP), User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) connections with low memory and flash consumption of approximately 11 kB according to [32].
Contiki has a flexible build and configuration system built on `make`. Many configuration options, implemented as C preprocessor definitions, can be changed by defining them directly while building, in the `Makefile` or in a custom `project-conf.h`.

2.3.1 Protothreads

Contiki implements lightweight event driven threads, called protothreads [12]. In comparison with a traditional multi-threaded architecture, these threads do not have a separate context-stack for each thread to save the current working state. This is important, as devices used in a WSN only provide a limited amount of memory. The protothreads extend the concept of a state machine, but provide additional C macros to simplify the implementation. Beside not using stacks, the threads are also non-preemptive, thus the application needs to cooperatively suspends its execution, so that the scheduler can activate another thread. The Listing 2.1 show an example protothread which prints `Hello World!\n"` every 1 s. It starts by creating a new Contiki process called `hello_world_process` that is added to the list of autostarted processes. The processes in this list get started from the main automatically when the system boots, thus no other file needs to be modified. The process creates a `static` variable `et` to store the state of a timer (`etimer`) that triggers an event after its scheduled time has passed. Defining the variable as `static` is important to not create it on the local call stack which is discarded upon waiting for the event.
The main parts of this protothread are:

- `PROCESS_BEGIN()`: it marks the beginning of a protothread.
- `PROCESS_WAIT_EVENT_UNTIL()`: it is used to wait for an event to happen. This exits the function and returns to the kernel/scheduler, which is then able to processes another process. If an event is triggered the execution continuous after this function. There exist also other function which "yield" and return back to the kernel/scheduler.
- `PROCESS_END()`: it marks the end of a protothread.

[10] http://www.contiki-os.org
[11] https://github.com/contiki-os/contiki

```
1  PROCESS(hello_world_process, "Hello world process");
2  AUTOSTART_PROCESSES(&hello_world_process);
3
4  PROCESS_THREAD(hello_world_process, ev, data)
5  {
6    PROCESS_BEGIN();
7    static struct etimer et;
8
9    while(1) {
10     etimer_set(&et, (CLOCK_SECOND));
11     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
12     printf("Hello world!\n");
13   }
14
15   PROCESS_END();
16 }
```

Listing 2.1: Print "`Hello World!\n`" every 1 s.

```
1  static char process_thread_hello_world_process(struct pt *process_pt,
       ↪ process_event_t ev, process_data_t data);
2  struct process hello_world_process = { NULL, "Hello world process",
       ↪ process_thread_hello_world_process };
3  struct process * const autostart_processes[] = {&hello_world_process, NULL};
4
5  static char process_thread_hello_world_process(struct pt *process_pt,
       ↪ process_event_t ev, process_data_t data)
6  {
7    char PT_YIELD_FLAG = 1;
8    if (PT_YIELD_FLAG) {;}
9
10   switch((process_pt)->lc) {
11     case 0:
12       static struct etimer et;
13
14       while(1) {
15         etimer_set(&et, (CLOCK_SECOND));
16
17         PT_YIELD_FLAG = 0;
18         (process_pt)->lc = __LINE__; case __LINE__:
19         if((PT_YIELD_FLAG == 0) || !(etimer_expired(&et))) {
20           return PT_YIELDED;
21         }
22
23         printf("Hello world!\n");
24       }
25   };
26
27   PT_YIELD_FLAG = 0;
28   (pt)->lc = 0;
29   return PT_ENDED;
30 }
```

Listing 2.2: “Hello-World” application with expanded protothread macros.

These protothread macros simplify the implementation. To better understand the principle of using a `switch()` statement to control program flow, another version of this example is created. Listing 2.2 shows a version where the macros are expanded.
The `(process_pt)->lc` variable is used for so called "local-continuation". This exploits the fact that C allows "jumping" in and out of other blocks like `while` and `if` using the `switch()`/`case` statement. The `PROCESS_WAIT_EVENT_UNTIL` macro remembers the current code line (`__LINE__`) and returns from the function if the condition is not yet met. The next time this thread executes, the execution continues at the previously saved line by switching to the corresponding `case` statement and checking the condition again.
This behaviour also makes it clear that variables defined at the local call stack are discarded upon calling a `WAIT` function, as this effectively returns from the function. Special care needs to be taken when using `switch()` statements in a protothread, as it is not allowed to call any protothread related wait function inside of another `switch()` as a new `case` would be generated for the nested `switch()` statement instead of the intended one. As the local continuation variable `lc` in Contiki uses a 16 bit type, the memory overhead for each thread is only 2 bytes plus some storage in the management list of all processes.

2.3.2 Energest

To assist in the development of low-power systems, Contiki includes a software based energy estimator (Energest). This helps to determine in which parts of the application code most energy was spent. The principle is to count the time spent while a certain software or hardware module is active and then calculate the energy and power using known constants for the corresponding current usage. As the current consumption of each component was only measured once, Energest only gives a rough estimation compared to really measuring the energy using a professional hardware device. The results, however, still allow a comparison between different experiments and give a quantitative idea of the energy-efficiency of a program.
For this thesis, the following Energest counters and current reference values are of interest:

- `LISTEN` - 20.0 mA
  The time spent while listening for and receiving a packet.
- `TRANSMIT` - 17.7 mA
  The time spent transmitting a packet.
- `CPU` - 1.8 mA
  The time spent while the CPU is active.
- `LPM` - 0.0545 mA
  The time spent while the CPU is idle and in low power mode.

The values are only valid for the Tmote Sky nodes used. To calculate the electrical power also the operating voltage needs to be known, the typical voltage of a Tmote Sky is 3.3 V. The Energest statistics are enabled by default (`ENERGEST_CONF_ON`), to access them the easiest method is to print the values periodically. The `print_stats()` from `print-stats.c` is already provided and just needs to be called periodically. In the example in Listing 2.3 the statistics are printed every 10 s.

```
E 0.18 clock 10 cpu 22668 lpm 306722 irq 16578 gled 0 yled 0 rled 0 tx 0
    ↪ listen 7538 sensors 0 serial 0
E 0.18 clock 20 cpu 60966 lpm 596104 irq 32466 gled 0 yled 0 rled 0 tx 1934
    ↪ listen 16464 sensors 0 serial 0
E 0.18 clock 30 cpu 86600 lpm 898150 irq 48918 gled 0 yled 0 rled 0 tx 2053
    ↪ listen 23839 sensors 0 serial 0
```

Listing 2.3: Sample output of Energest every 10 s.

Each line starts with an `E` followed by two bytes of the nodes link address. Thereafter follows the current system runtime in seconds and then a counter value for each measured energy category. These counter values are determined using Contiki's fine grained `RTIMER`, where one tick typically corresponds to 30.5 µs (RTIMER_SECOND = 32,768 Hz).
To determine the average power consumption in the last 10 s, first the counter values need to be subtracted. The sum of `CPU` and `LPM` should then correspond to 10 s whereas the other counter values indicate the corresponding time the function was active during this period.

$$t_{\texttt{LISTEN}} = t_{\texttt{LISTEN}_2} - t_{\texttt{LISTEN}_1} = (23839 - 16464)/32{,}768\,\text{Hz} = 0.225\,\text{s} \tag{2.1}$$
$$t_{\texttt{TRANSMIT}} = 0.004\,\text{s} \tag{2.2}$$
$$t_{\texttt{CPU}} = 0.782\,\text{s} \tag{2.3}$$
$$t_{\texttt{LPM}} = 9.218\,\text{s} \tag{2.4}$$
$$t = t_{\texttt{CPU}} + t_{\texttt{CLPM}} = 10\,\text{s} \tag{2.5}$$

After the time of each component is computed, the next step is to estimate the energy and the average power for this time period.

$$E_{\texttt{LISTEN}} = t_{\texttt{LISTEN}} \cdot 20.0\,\text{mA} \cdot 3.3\,\text{V} = 14.85\,\text{mJ} \tag{2.6}$$
$$E_{\texttt{TRANSMIT}} = t_{\texttt{TRANSMIT}} \cdot 17.7\,\text{mA} \cdot 3.3\,\text{V} = 0.21\,\text{mJ} \tag{2.7}$$
$$E_{\texttt{CPU}} = t_{\texttt{CPU}} \cdot 1.8\,\text{mA} \cdot 3.3\,\text{V} = 4.65\,\text{mJ} \tag{2.8}$$
$$E_{\texttt{LPM}} = t_{\texttt{LPM}} \cdot 0.0545\,\text{mA} \cdot 3.3\,\text{V} = 1.66\,\text{mJ} \tag{2.9}$$

$$P_{\texttt{LISTEN}} = E_{\texttt{LISTEN}}/t = 1.49\,\text{mW} \tag{2.10}$$
$$P_{\texttt{TRANSMIT}} = 0.02\,\text{mW} \tag{2.11}$$
$$P_{\texttt{CPU}} = 0.46\,\text{mW} \tag{2.12}$$
$$P_{\texttt{LPM}} = 0.17\,\text{mW} \tag{2.13}$$

When looking at the results, they show that a lot of energy is used while listening on the radio, whereas actually transmitting is not so much. Energest allows to get a good overview where energy is spent and will be used in this thesis to compare the efficiency of the protocols and applications under different conditions, such as radio interference intensities.

2.3.3 Networking stack

The Contiki networking stack as shown in Figure 2.6 consists of multiple layers. Each of them can be configured to a different implementation by changing a `C` preprocessor define.

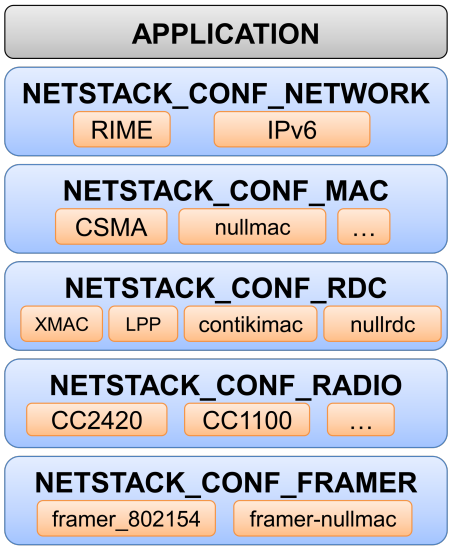


Figure 2.6: The Contiki networking stack (Source: LAC-Labor Lecture Slides).

The layers are as follows:

- Application Layer
  The application layer sits on top of the other layers and is implemented by the user.
- Network stack - `NETSTACK_CONF_NETWORK`
  This thesis uses the IPv6 network protocol, which also includes 6LoWPAN and RPL.
- MAC protocol - `NETSTACK_CONF_MAC`
  This layer defines the used MAC protocol. This thesis uses the default carrier sense multiple access (CSMA) implementation which handles multiple retransmissions with an exponential backoff to better avoid collisions.
- radio duty-cycling (RDC) protocol - `NETSTACK_CONF_RDC`
  The radio duty-cycling protocol used in this thesis is `ContikiMAC` (described in Section 2.4.3).
- Radio driver - `NETSTACK_CONF_RADIO`
  The Tmote Sky has a CC2420 radio, so the CC2420 radio driver is used.
- Packet format conversion - `NETSTACK_CONF_FRAMER`
  As this thesis operates on a IEEE 802.15.4 network, also the packet framer for this protocol is used.

Each layer provides a set (`struct`) of callback functions. The layers then call the corresponding functions to achieve the desired result.

2.3.4 Working with Contiki

To start working with Contiki the latest source code can be obtained from GitHub using `git` and the command in Listing 2.4.

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

Listing 2.4: GIT clone of the contiki repository.

After this step, compiling and uploading an example application can be performed as follows. To successfully compile for the Tmote Sky a MSP430 `gcc` is needed. On macOS, the simplest way to install an MSP430 compiler is using a Homebrew[12] formula. Homebrew is a package manager designed for macOS which simplifies installation of various command line tools. The Listing 2.5 installs the `msp430-gcc` which is then used by Contiki OS.

```
$ brew install sampsyo/mspgcc/msp430-libc && \
  $(brew --prefix)/Library/Taps/sampsyo/homebrew-mspgcc/addlinks.sh
```

Listing 2.5: Install the `msp430-gcc` using `brew`.

To build the IPv6 RPL UDP example the commands in Listing 2.6 can be used. First change to the corresponding example folder and then execute the `make` command.

```
$ cd contiki/examples/ipv6/rpl-udp
$ make TARGET=sky PERIOD=10
```

Listing 2.6: Building the `rpl-udp` example.

After this step, the build system creates two firmware images: (`udp-client.sky`) and (`udp-server.sky`). The UDP client sends a packet every 10 s to the UDP server. The server also functions as RPL root starting to create a routing tree. Before the executables are uploaded to the nodes, the command in Listing 2.7 allows to determine which nodes are connected to the developer's computer.

```
$ make TARGET=sky motelist
../../../tools/sky/motelist-macos
Reference  Device                       Description
---------- ---------------------------- ----------------------------------
FTYLN1ME   /dev/tty.usbserial-FTYLN1ME (none)
FTYLN2UP   /dev/tty.usbserial-FTYLN2UP (none)
FTYLN2XL   /dev/tty.usbserial-FTYLN2XL (none)
FTYLN3EE   /dev/tty.usbserial-FTYLN3EE (none)
```

Listing 2.7: Printing a list of connected nodes.

Using the `*.upload` target of the `Makefile` the executable can be uploaded to the connected nodes. First the client application is uploaded to all nodes. In a second step, the server is uploaded to the first node. The last command of the Listing 2.8 starts the `serialdump` utility to display the received log messages from the first node. This will then display the received messages at the server send from the other three nodes.

```
$ make TARGET=sky udp-client.upload
$ make TARGET=sky udp-server.upload MOTE=1
$ make TARGET=sky login MOTE=1
```

[12] http://brew.sh

```
...
DATA recv 'Hello 10 from the client' from 222
DATA recv 'Hello 11 from the client' from 222
DATA recv 'Hello 12 from the client' from 222
```

Listing 2.8: “Login" to a node using the `serialdump` utility.

2.4 Protocols used in this thesis

This section describes the protocols used in this thesis. In Section 2.4.1 an introduction to IPv6 and 6LoWPAN is provided, the Section 2.4.2 introduces the routing protocol for low-power and lossy networks (RPL). In Section 2.4.3, ConikiMAC, Contiki OS's default RDC protocol that is also used by in this thesis is presented.

2.4.1 IPv6 and 6LoWPAN

Using IP communication on a sensor node is a good idea as this is a well known and in general, on conventional PCs, good supported protocol. It also removes the need of a protocol translator if the node wants to store data to an IP connected server [32]. As the number of internet connected devices is rapidly growing and the IPv4 address space is quite limited, using IPv6 as a base for the nodes communication will solve this problem by using 128 bit addresses.
To get a basic understanding of IPv6 looking a the protocol header in Figure 2.7 is a good starting point.

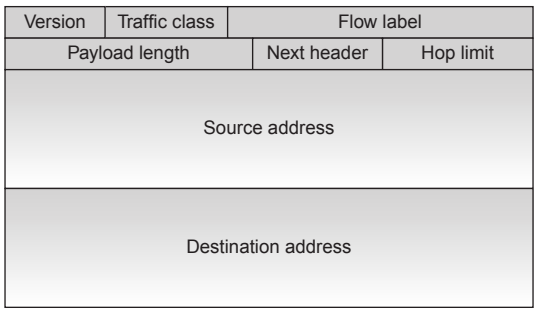


Figure 2.7: IPv6 packet format (Source: [32]).

The packet header has a length of 40 bytes consisting of multiple fields. The version field (4 bit) has for IPv6 always the value 6. The traffic class (8 bit) is used to offer a different quality of service (QoS) to some types of traffic, such as to prioritized Voice over IP (VoIP) communication. The flow label (20 bit) is still a mostly experimental flag to indicate a sequence of packets that require a specific handling. The payload length (16 bit) indicate the number of bytes in the payload, but excluding the length of the header. The next header field (8 bit) indicates the header type that immediately follows the IPv6 header. This allows for a flexible daisy chaining of optional extended headers such as routing, hop-by-hop options or authentication. The last next header field in a chain would then indicate the protocol of the payload such as TCP, UDP or ICMP. The next field is the hop limit (8 bit) which is decremented each time a nodes forwards the packet. Upon reaching 0 the

packet is discarded. The last two field are the source and destination addresses (128 bit each) which contain the IPv6 addresses of the sender or destination, respectively.
IPv6 mandates the support for a maximum transmission unit (MTU) of 1280 bytes. As the MTU of an IEEE 802.15.4 link is 127 byte (practically even less due to the MAC header overhead), implementing IPv6 inevitably leads to the need of packet fragmentation and reassembly. An IPv6 header with 40 bytes would also use a large portion of the limited payload size. To still support IPv6, the IETF started to work on an adaptation layer called 6LoWPAN which is specified in RFC 4944[13]. This layer is especially designed for IEEE 802.15.4 links as it enables a strong header IPv6 header compression by eliminating redundant informations. The three main services which are provided by 6LoWPAN are packet fragmentation and reassembly, header compression and link layer forwarding when using multiple hops. The 6LoWPAN encapsulation header stack is shown in Figure 2.8.

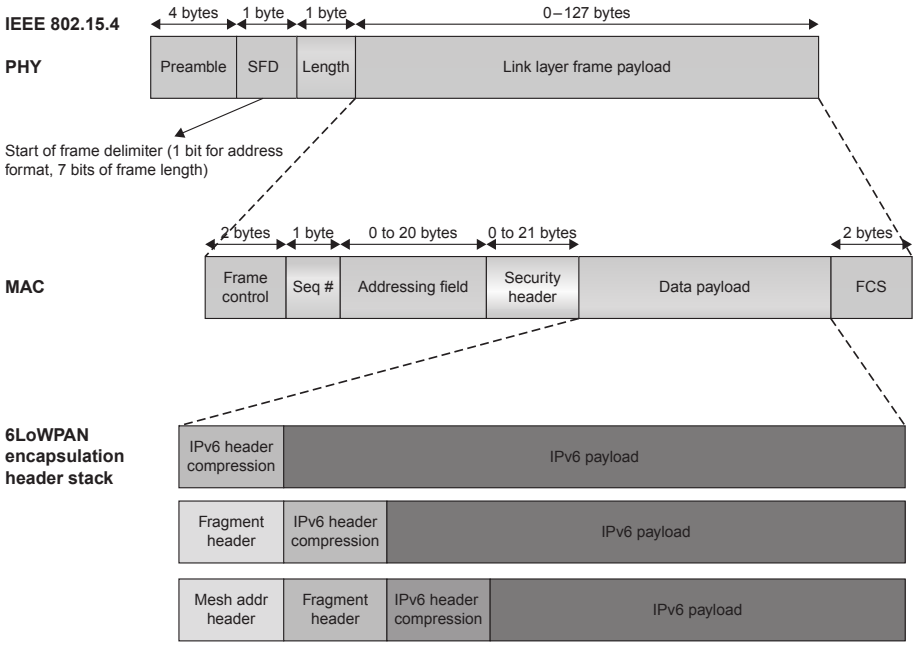


Figure 2.8: 6LoWPAN encapsulation header stack (Source: [32]).

To compress the IPv6 header, some field with known or unused values such as the version, traffic class and flow label can be easily omitted. Other fields such as the payload length can be removed as they can be reconstructed from the IEEE 802.15.4 packet length field. Reducing the source and destination address is possible by using the MAC layer addressing field and combining it with a prefix. The specification for the latest header compression called `LOWPAN_IPHC` can be found in RFC 6282[14].
The Contiki implementation of 6LoWPAN was originally developed as part of the SICSlowpan project at the Swedish Institute of Computer Science from August 2008 to April 2009. The aim of the project was to develop the first open source implementation of IPv6 of the low-power IEEE 802.15.4 radio layer [10]. The resulting implementation of the uIPv6 stack and SICSlowpan can be found in `core/net/ipv6`.

[13] https://tools.ietf.org/html/rfc4944
[14] https://tools.ietf.org/html/rfc6282

2.4.2 RPL - Routing Protocol for Low Power and Lossy Networks

To forward messages from individual nodes to a central sink, over multiple hops, a routing protocol needs to be employed. The traditional protocols used in the internet backbone are not suitable for a WSN. Packet drops on lossy links, as prevalent in WSN, are very likely, failures due to interference are often only transient. The sleeping behaviour and limited bandwidth of a typical node also requires special attention.
To work on routing in WSNs the IETF formed a working group called Routing Over Low-power and Lossy networks (ROLL) with the latest aim to develop and specify a new routing protocol called routing protocol for low-power and lossy networks (RPL). The specification for this protocol can be found in RFC 6550[15]. It incorporates the special characteristics found in a WSN.
RPL is a distance-vector algorithm that builds a destination oriented directed acyclic graph (DODAG). A graph in which all the paths have a direction and do not form any cycles is called directed acyclic graph (DAG). A DODAG is a special form if there exists a path from every node to one special node called sink or DODAG root.

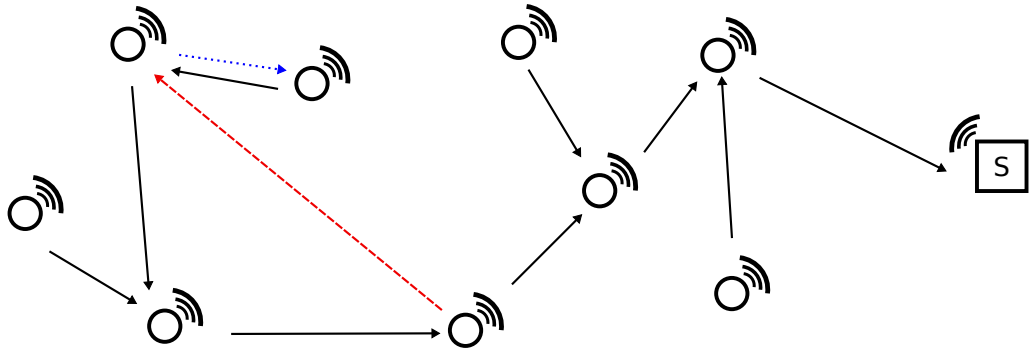


Figure 2.9: A example RPL DODAG is created by using the black solid paths.

Figure 2.9 shows an example of a DODAG if only the black solid paths are used. The red dashed path forms a cycle which breaks the DAG requirement. The direction of the blue dotted path is valid for a DAG but the direction needs to be changed to get a valid DODAG as otherwise this node would not have a path to the DODAG root.
To build this graph, RPL defines three additional ICMP messages. The DODAG information object (DIO) is broadcasted by a node periodically and contains the nodes rank and other DODAG characteristics. A node may solicit a DIO by sending a DODAG information solicitation (DIS) message to its neighbours. This is useful for newly activated nodes to quickly join the DODAG as the periods of the DIS messages increases as long as the graph is stable. The third message is the destination advertisment object (DAO) and it is sent from a node to its parent (ancestor) to populate their routing tables with the informations gathered by other DIO and DAO messages. To describe its logical position in the DODAG each node has a *rank*. The rank decreases the closer a node is to the RPL root, which has a rank of 0. This rank is primarily used for loop-avoidance, as only selecting another node with a rank smaller than its own is allowed. The rank of a node is derived from a routing *metric*. The metric calculation is performed by an *objective function*. An objective func-

[15] https://tools.ietf.org/html/rfc6550

tion (OF) combines multiple metrics and constraints to find a minimum cost path. The simplest OF tries to minimize the hop count, it is defined as objective function 0 (OF0) in `rpl-of0.c`. The other metric used in this thesis is based on expected transmission count (ETX). This OF minimizes the number of transmissions to reach the sink, this is similar to OF0 but also considers the number of transmission between two nodes as an added link metric. The Contiki OS implementation of an ETX based OF is called *minimum rank with hysteresis objective function* and is implemented in `rpl-mrhof.c`.

2.4.3 ContikiMAC

`ContikiMAC` is the default radio duty-cycling (RDC) protocol used in Contiki OS [11]. As this thesis focuses on the routing layer only, the default protocol is used.
Radio duty-cycling is a method to reduce the energy consumption of a node by turning the node's radio on only periodically and keeping it off the rest of the time. To still be able to communicate with other nodes, every node needs to operate using the same RDC protocol with the same parameters. In a basic implementation as shown in Figure 2.10, the receiver has a periodic reception window during which the radio is turned on. If the sender wants to transmit a packet `D`, it repeatedly transmits `D` as long as no acknowledgment (ACK) `A` has been received. If `A` has not been received for a period longer than the reception interval, the transmitter stops transmitting and the packet is marked as lost. The CSMA-MAC layer will then care of resending.

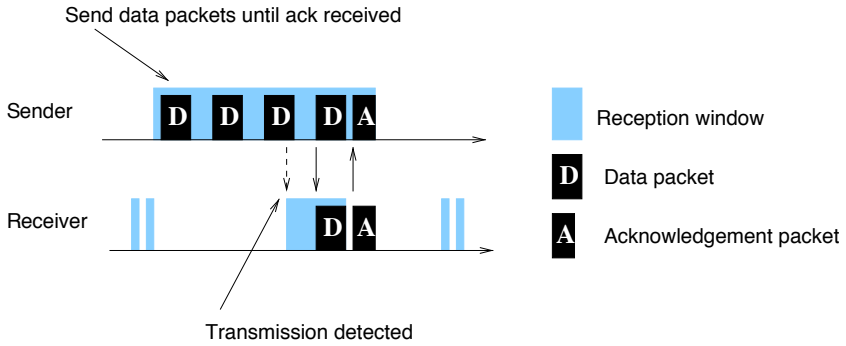


Figure 2.10: Basic communication schema of `ContikiMAC` protocol (Source: [11]).

The protocol does not use any special strobing packets or agreements as in X-MAC [6]; the payload is sent directly and link layer ACKs are used to indicate a successful transmission in order to simplify the implementation.
In `ContikiMAC`, to minimize the radio-on-time, after turning the radio on, a fast CCA check is performed. If there is something in the air, the radio is kept on for a longer period, as there might be some data coming in. In the current implementation the threshold at which a signal is considered is fixed just above the noise floor.

Phase lock

To further minimize the energy consumption, `ContikiMAC` implements a phase lock mechanism. By keeping track of the time when a packet was sent successfully, a node learns the

wakeup interval of the other node. This allows the node to start sending just before the reception window of the other node as shown in Figure 2.11.

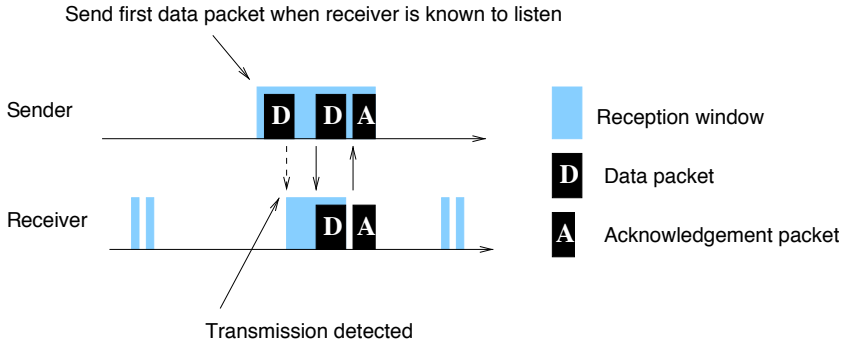


Figure 2.11: `ContikiMAC` with enabled phase lock (Source: [11]).

This mechanism works quite well, but also requires careful timing. If there are longer periods in which a nodes is not sending, their clocks might drift apart and fall out of synchronization. This might lead to higher energy consumptions as more packets have to be transmitted.
The current implementation of the phase lock mechanism in `phase.c` is also quite sensitive to added debug output using `printf`. During development some added debug statements led to a complete internal lock up, as no messages were sent. When sending a message the implementation calculates in the `phase_wait()` function, the time needed to wait until the receiver wakes up. Depending on the time needed to wait, the code will either wait in a busy-loop until the correct time is reached or "defer" sending the message using a callback timer (`ctimer`). After this timer expires a new attempt to send the message is performed, calculating again the time needed to wait. If this timer fires too late, the calculation may defer the sending again as the current wakeup cycle is missed. Firing too late may happen due to a longer execution time either while calculating the needed waiting timer or between calculating the next timeout value and actually starting the timeout.
Adding additional debug statements may lead the code to "defer" always, as the next cycle will always be missed. Missing cycles and deferring more than once also happen without added debug statements but seldom more than twice. This does not cause a lock up but increases the latency. As a solution to this problem incorporating the execution time in the calculation was tried however this did not yield the expected results.

Clear channel assessment

The CCA in Contiki OS is used both when sending and also when receiving a message. It is always used if there is the need to know if the radio channel is occupied by any other sender. The actual assessment is performed by the `CC2420` radio chip. The radio chip sets the CCA pin if the measured received energy (RSSI) is below a configurable threshold (`RSSI.CCA_THR`).
Before sending a message the CCA pin is checked to assure the channel is free. On the other hand `ContikiMAC` checks the CCA pin with the defined Channel Check Rate (CCR) to determine if anything receivable might be in the air. If a signal above the CCA threshold is detected the radio is kept on for a longer period to either receive the current packet or the next retransmission. The `CC240`'s default value for this threshold is $-77\,\mathrm{dBm}$.

2.5 Radio interference

In general, interference is called the situation in which an undesired effect negatively affects the operation of a device. This thesis deals only with radio interference where the reception of a radio signal is badly affected by another signal. Other environmental effects interfering with WSNs such as temperature or humidity are not considered.
The ability to receive a message correctly depends on the strength of the received signal. The required strength of the desired signal above the unwanted signal is defined as “Co-channel rejection". For the CC2420 radio this value is defined as 3 dB. This means, in order to successfully receive a message, its signal strength needs to be 3 dB above the interference or the noise floor. This thesis uses the term “noise floor" to the define the received signal strength of the radio if the is no communication ongoing and thus indicates the sensitivity of the radio.

2.5.1 Generating interference

To generate interference the CC2420 on Tmote Sky nodes itself can be used. In 2011 a tool called JamLab was developed to allow the generation of realistic and repeatable interference using the test modes of the CC2420 radio [3]. Using Tmote Sky nodes itself is a quite novel approach, as it eliminates the need to bring the real interferers into a testbed and to configure them remotely.
The source to this Contiki application was provided and used to emulate various interference patterns. The provided patterns are:

- `JL_NOINT`: No interference
- `JL_MICRO`: Microwave oven
- `JL_BLUET`: Bluetooth
- `JL_WIFI1`: Wi-Fi radio streaming
- `JL_WIFI2`: Wi-Fi video streaming
- `JL_WIFI3`: Wi-Fi file transfer
- `JL_WIFI4`: Wi-Fi file transfer and radio streaming
- `JL_CONTN`: Continuous carrier

The Bluetooth and Wi-Fi (802.11b) patterns use prerecorded probabilities where the Jammer is active, while the microwave oven is modelled using a periodic jammer with a certain period and duty-cycle. The continuous carrier mode turns on an unmodulated carrier for the whole time. This mode is used to understand how strong each node is affected by the generated noise with a probability of 100 %.
To enable an unmodulated carrier the radio needs to be configured by writing certain test registers (`DACTST` to 0x1800, `MANOR` to 0x0100, `MDMCTRL1` to 0x0508, `TOPTST` to 0x0004 and `STXON`). After this, it is possible to turn the carrier on and off by changing the transmission power.

In this thesis a JamLab interferer example is used, which allows the configuration of an jamming “ON" and a jamming “OFF" periods and the jamming pattern at compile time. To test with multiple configurations, separate test programs need to be build and uploaded to the nodes.
There are some things to be considered: as the RF output power of a CC2420 radio is much smaller than those of a Wi-Fi access point (AP) or a microwave oven, directly replacing a Wi-Fi AP by a Tmote Sky will influence a much smaller area. This can either be mitigated by using multiple nodes or by assuming the Wi-Fi AP is far away. Another thing is that the CC2420 radio can only communicate on a single channel and thus also only interfere on a single narrow band spectrum, whereas a microwave oven has a much broader spectrum. This fact can be neglected as the typical communication and also the communications used in this thesis operate only on a single channel.

2.5.2 Measuring interference

To measure the noise, one can read the received signal strength indicator (RSSI). It can be read any time as long as the radio is turned on and it is automatically read after a packet has been received. To determine the noise floor a high number of measurements using the `noise_analyzer` are performed. These measurements are then preprocessed using a histogram-like data structure directly on the node. This reduces the data rate needed to transfer all measurements using the debug interface. Each read RSSI value gets counted at its corresponding “bin". After the desired number of values is measured the list gets printed as shown in Listing 2.9. In this example 73 % (14,603) of the 20,000 readings were at −95 dBm.

```
Scanning for noise and interference. Estimated waiting time between
    ↪ iterations: 640 seconds...

Noise on channel 26:
-100:       0 (0)
 -99:       0 (0)
 -98:       0 (0)
 -97:       0 (0)
 -96:      28 (28)
 -95:   14603 (14631)
 -94:    5369 (20000)
 -93:       0 (20000)
 -92:       0 (20000)

...
```

Listing 2.9: Noise floor measurement on a noide on channel 26.

Reading the value directly from the radios `RSSI.RSSI_VAL` register returns an 8 bit signed 2’s complement integer [31]. To get the power at the RF pins another offset needs to be added to this value. This `RSSI_OFFSET` can be found empirically, but the datasheet states an approximate value of −45, which is used in this thesis. In recent version of Contiki this value (45) is added to the readings automatically, but attentions needs to be paid to some older example which may not yet incorporate this change and then report false readings.

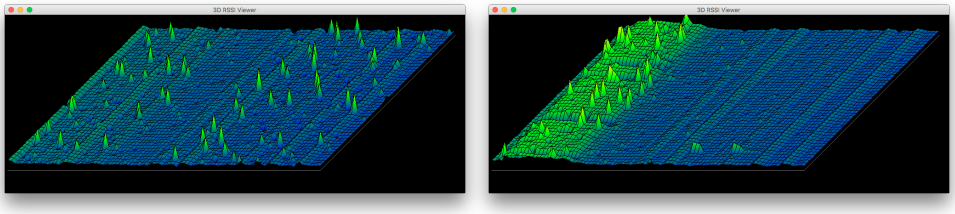
$$P = \texttt{RSSI\_VAL} + \texttt{RSSI\_OFFSET}\,[\text{dBm}] \tag{2.14}$$

According to the datasheet the RSSI has a dynamic range of about 100 dB from −100 dBm to 0 dBm, an accuracy of ±6 dB and a linearity of ±3 dB.
To quickly visualize the current radio spectrum the `rssi-scanner` Contiki example can be used. This Contiki application quickly measures the RSSI of the supported spectrum and also allows passing the collected data to a Java application which then visualizes the data either in 2D or in 3D.

```
$ cd examples/rssi-scanner
$ make TARGET=sky rssi-scanner-cc2420.upload
$ make TARGET=sky viewrssi3d
```

Listing 2.10: Upload the `rssi-scanner` example and start the 3D visulization.

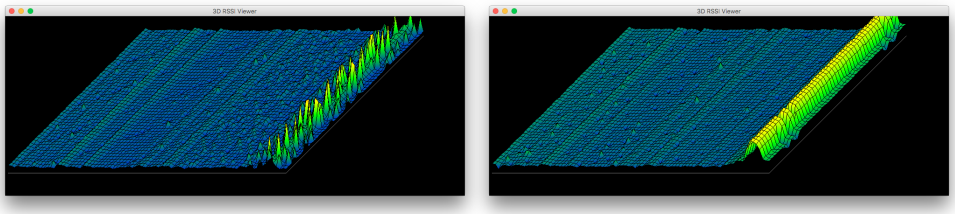
Example data visualizations are shown in Figure 2.12. The 3D RSSI viewer shows 86 distinct measurements from 2.400 GHz to 2.485 GHz including the history on a third axis. Figure 2.12a shows an ongoing Bluetooth communication, where the channel hopping is clearly visible. In Figure 2.12b a Wi-Fi file stream on the Wi-Fi channel 1 is monitored.



(a) Bluetooth (b) Wi-Fi 802.11g on channel 1

Figure 2.12: 3D visualization of the RSSI of Bluetooth and Wi-Fi.

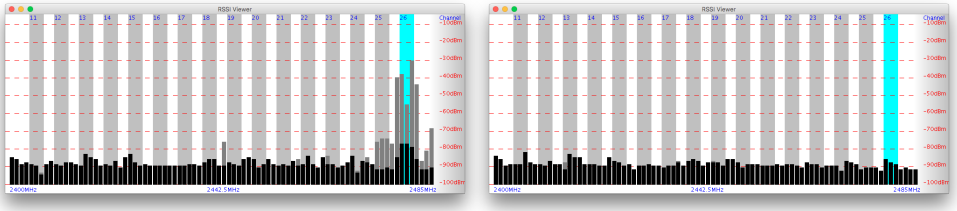
This thesis uses JamLab as described in Section 2.5.1 and so measurements while generating noise using JamLab on a Tmote Sky node are of interest. Figure 2.13 shows emulated interference on channel 26.
The Figure 2.13a shows an emulation of a Wi-Fi file and radio streaming communication (`JL_WIFI4`) using 802.11b, whereas Figure 2.13b depicts a continuous enable carrier.



(a) Emulated heavy Wi-Fi (b) Emulated continuous carrier

Figure 2.13: Emulated signals by using JamLab.

Using the 2D visualization of the RSSI scanner as shown in Figure 2.14 it is possible to get a feeling for the actual RSSI values. The emulated Wi-Fi in Figure 2.14a creates a radio signal with around −50 dBm. While no interference is present, the node measures the noise floor, which indicates the lowest receivable signal strength. For the CC2420 this values is around −90 dBm as also visible in Figure 2.14b.



(a) Emulated heavy Wi-Fi on channel 26 (b) Noise floor without interference

Figure 2.14: RSSI measurements visualized using the 2D RSSI viewer.

2.5.3 Tracing packets through the network

Following a single packet through the network is not easily possible without modifications. Logging each forwarded packet at each node also does not guarantee the traceability, as it is hard to combine this information. By extending the RPL IPv6 header to include every node the packet travelled as shown in Listing 2.11, it is possible to trace the path of every single packet. The full list of nodes the packet has passed is printed on every hop. This allows to even trace lost packets to their last successful hop.

```
typedef struct uip_ext_hdr_opt_rpl {
     uint8_t opt_type;
     uint8_t opt_len;
     uint8_t flags;
     uint8_t instance;
     uint16_t senderrank;
     #if RPL_ENABLE_TRACEROUTE
     uint16_t seq_no;
     uint8_t visited_nodes[RPL_VISITED_NODES_BUFFER];
     int8_t rssi[RPL_VISITED_NODES_BUFFER];
     #endif
} uip_ext_hdr_opt_rpl;
```

Listing 2.11: Extending the RPL header to include tracing data.

The traces also include the RSSI of the packet as it was received: this should help identifying whether the connection between the nodes (and thus the parent selection) was good. It also contains a per node unique 16 bit sequence number that aids when parsing the logs of each individual node and to find the longest sequence of hops.
The RPL header is processed by each hop, making it ideal to add this extension. The length of the header must be a multiple of 8 as stated by the RFC 2460[16] the length field of the Hop-By-Hop options header uses a 8-octet unit. Changing only the `RPL_HDR_OPT_LEN` was not sufficient as some bugs in Contiki prevented a value greater than 8 (See this PR[17]).

[16] https://tools.ietf.org/html/rfc2460#section-4.3
[17] https://github.com/contiki-os/contiki/pull/1687

A partly working version of this tracing functionality has been provided and was integrated into the used Contiki source code version.
The debug output of trace at the sink (108) looks as shown in Listing 2.12. The messages start with the sequence number, followed by the originating node and listing the next hops including the RSSI in braces.

```
RPL: travelled 146: 127 (0), 100 (-61), 108 (-80)
RPL: travelled 145: 110 (0), 108 (-76)
RPL: travelled 145: 106 (0), 108 (-70)
RPL: travelled 145: 131 (0), 127 (-77), 100 (-61), 108 (-80)
RPL: travelled 147: 112 (0), 108 (-73)
RPL: travelled 146: 125 (0), 106 (-78), 108 (-70)
RPL: travelled 145: 101 (0), 102 (-75), 112 (-75), 108 (-73)
RPL: travelled 147: 100 (0), 108 (-80)
RPL: travelled 147: 116 (0), 100 (-71), 108 (-80)
RPL: travelled 146: 126 (0), 112 (-76), 108 (-72)
RPL: travelled 146: 115 (0), 106 (-75), 108 (-70)
```

Listing 2.12: Debug output of the RPL trace functionality.

Chapter 3

Experimental Campaign

A big part of this Thesis involves experimentally evaluating RPL under realistic environmental conditions. The experiments should also show where problematic points are and whether they can be mitigated.
Multiple properties of the environment affect the radio communication, such as temperature, humidity, and radio interference. The experiments in this Thesis focus on the influence of radio interference on RPL in general and more specifically on the CCA used by `ContikiMAC` to determine if other nodes are currently sending messages.
RPL is designed to be used in lossy networks. Even with high packet loss on the physical layer, it should be possible to create a stable network. The following experiments try to confirm this assumption.
Other researchers have already shown that RPL might perform poorly under heavy interference and also suggest better coordination between network layers [15]. The experiment in prior work were either run in an emulator such as Cooja, using other objective function, or on a different RPL implementation (e.g., TinyOS). This Thesis uses the latest Contiki OS RPL implementation on a real testbed.

The goals of the experiments described in this chapter are:

- Determine the stability and reliability of RPL in a real testbed under various conditions of radio interference.
- Radio interference conditions: None, Light Wi-Fi, Strong Wi-Fi, Constant Carrier.
- Compare metrics between different RPL parameters and network configurations.
- Identify problems in the communication and investigate them further.

To achieve these goals the following methods are used:

- Setup a network of about 30 nodes using Contiki OS's RPL implementation.
- Generate radio interference using JamLab as described in Section 2.5.
- Collect various metrics as defined in Section 3.1.
- Use existing tools or extend and implement new tools to automate experiments.

This chapter describes the metrics in Section 3.1 and the setup used in Section 3.2 to test RPL in a real testbed while artificial, but realistic, radio interference is present. This chapter then continues in Section 3.2.5 describing all the created and used tools to achieve the results presented starting with Section 3.3.

3.1 Evaluation Metrics

During an experiment several metrics are collected. These metrics should indicate if a configuration is performing better than another. The following list gives an overview of the considered and collected metrics:

- *packet reception ratio*
  The PRR is calculated by dividing the number of messages received at the sink by the number of messages sent. In this thesis the PRR is always calculated from sender to sink of UDP payload packets, ignoring ICMP control and forwarded packets. A higher PRR indicates better performance.

- *Energy*
  A lower energy consumption indicates better performance, the energy consumption is measured using Energest as described in Section 2.3.2. The interesting counts are the CPU on time as well as radio listening and transmission times. The measurement should be accurate enough to compare multiple experiments.

- *Latency*
  Packet latency would also be an interesting metric, but it is ignored as it is hard to measure in the current testbed setup. There is no time synchronisation between the nodes only the PC time when writing to the log file is available.

- *Parent switches*
  A stable network has a low number of parent switches as "good" parents should be selected. Each (unnecessary) parent switch wastes energy and lowers the stability of the network. Packets might need to be retransmitted if the selected parent is not reachable. This metric records the number of parent switches performed. This metric is also of high interest while investigating the effects of a periodic jammer.

- *Control overhead*
  RPL uses special ICMP messages to communicate. Counting these messages and putting them in relation to the number of UDP packets gives an estimation of the control overhead. The trickle timer used in RPL should decrease the overhead as the network stabilizes. A jammer might destabilize the network and thus lead to a higher control overhead.

- *Hop count*
  Implementing a trace functionality as part of a 6LoWPAN extension header allows following packets throughout the network and also records the number of hops a packet travelled (one can use this to track down where a packet got lost). More details of the tracing functionality are described in Section 2.5.3.

- *RPL local and global repairs*
  Counting the number of local and global RPL DODAG repairs also provides some information to compare different configurations. A global repair is quite time consuming and should be avoided.

- *ContikiMAC statistics*
  Statistics about `ContikiMAC` are not used directly as a metric, but still collected to better understand the networks behaviour. The collected metrics as part of the `ContikiMAC` statistics are information about sent packets: timestamp, number of strobes, transmission result (lost, received ACK, collisions before sending, collisions while sending) and information about received packets: timestamp, length, RSSI.

3.2 Experimental Setup

The experimental setup runs multiple network configurations (RPL and RDC parameters, network size and radio interference patterns) to collect the results and metrics as described in Section 3.1.
These configurations are defined in shell scripts in the `Source/run-configurations/` folder. To start one test run the `Source/run-iti.sh` script is called. This scripts defines all active *run-configurations* and their run duration. The typical experiment length is between 30 minutes and one hour. Some of the scripts just measure the noise floor while the jammer is active using the methods described in Section 2.5.2. Most of the scripts run RPL in a network as shown in Figure 3.1 consisting of two to 30 nodes.
The built network consists of four types of nodes:
Most of the nodes are *Sender*-Nodes sending a UDP message to the single *Sink*-Node every 10 s with a random interval of 10 s. Each *Sender*-Node also forwards messages from other nodes, if required by the created RPL tree. The next type is the *NoiseAnalyzer*-Node. This node type continuously measures the noise floor.
The fourth type of nodes is the *Jammer*-Node. It creates radio interference using JamLab as described in Section 2.5.1. The tests use the following interference patterns typically present in a smart home application:

- light Wi-Fi (`JL_WIFI1`) emulating radio streaming.
- light Wi-Fi (`JL_WIFI2`) emulating video streaming.
- heavy Wi-Fi (`JL_WIFI4`) emulating file transfer and radio streaming.
- Continuous enabled carrier (`JL_CONTN`) emulating a Malicious Jammer completely saturating the channel.

All tests in which a jammer is used the latter is turned on after 5 minutes and kept enabled until the test finishes. This allows the RPL tree to be formed.
The transmission power used for the jammer is always 11 (−10 dBm) and for the rest of the nodes participating in RPL it is 4 (−22 dBm) (except where otherwise noted). These values were chosen empirically as shown in Section 3.2.1 and depend on the physical distances between the nodes as well as on the characteristics of the testbed room used. A single transmission power is used, as it is assumed that this is the highest available level.

The experiments mostly use `ContikiMAC` as it is Contiki's default RDC protocol, some experiments are also run using `nullrdc` as a comparison. The CCR used is 32 Hz: this frequency was chosen to better factor out packet losses due to congestion. Using a lower CCR decreases the energy consumption as shown in Section 3.2.2, but also increases the packet loss and latency. The CCR may be decreased if less nodes are used or less messages are sent.
The experiments were designed to run for 60 minutes: some of the simpler experiments, for example measuring the noise impact, run for a shorter period, as this is also sufficient to get meaningful results and reduces the overall testbed usage time.
The RPL application used is based on the `rpl-collect`[1] sample code and thus sends UDP packets with a 46 Byte payload. The payload contains some diagnostic data, which can be loaded into `collect-view`[2], as shown in Figure 3.1. The downside to use `collect-view` is that it only works on data collected by the sink.
The experiments are run using the ETX OF, but also some experiments were run using the OF0 to confirm that ETX yields better results in terms of PRR.

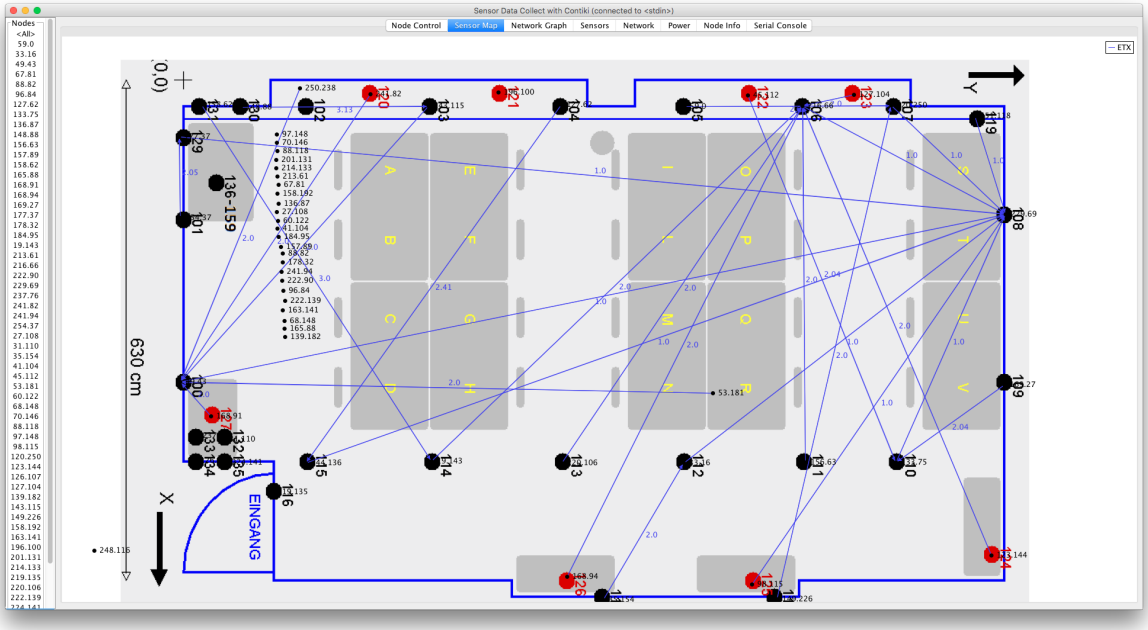


Figure 3.1: Node map visualized using `collect-view`.

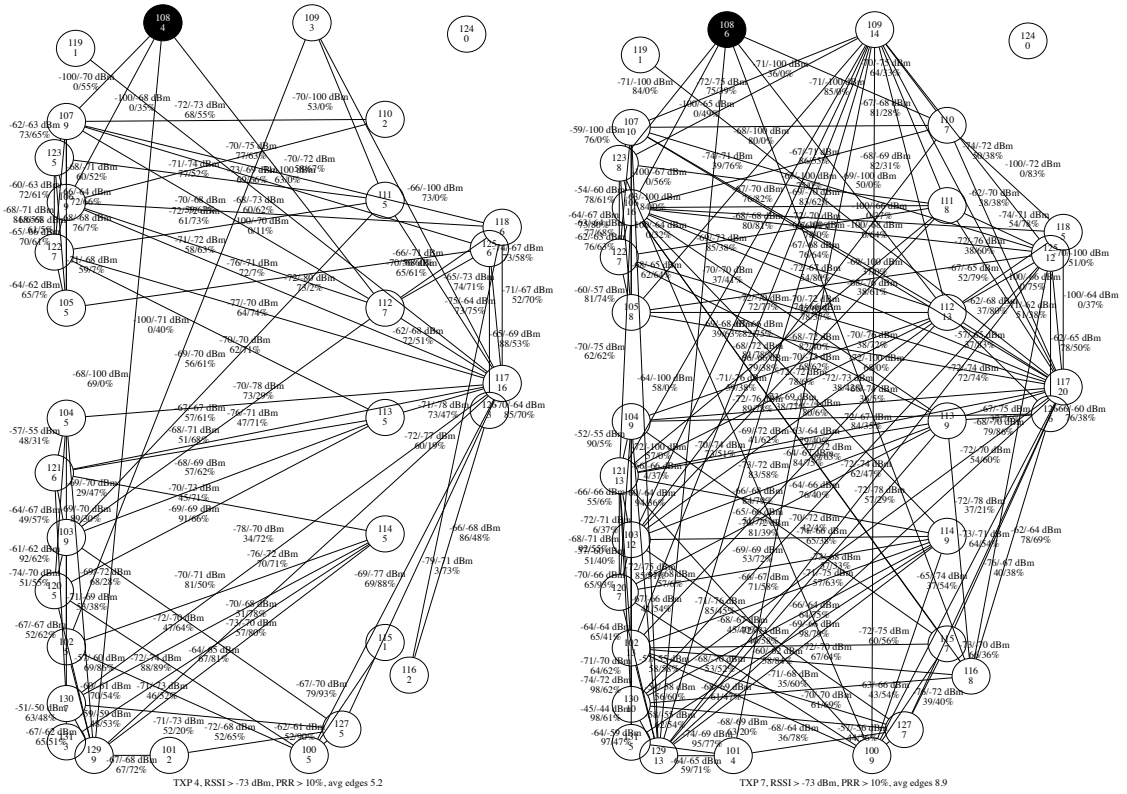
Determining the used transmission power is a task independently of RPL. How the transmission power was chosen is described in the following Section 3.2.1. Also choosing a sensible CCR is important, it depends on the expected packet send rate and the allowed latency and has also an influence on the power consumption. As the CCR is not varied between experiment, the Section 3.2.2 describes how the power consumption changes between different values.

[1] https://github.com/contiki-os/contiki/tree/master/examples/ipv6/rpl-collect
[2] https://github.com/contiki-os/contiki/tree/master/tools/collect-view

3.2.1 Determining the transmission power

Selecting a good transmission power (TXP) for the WSN is not an easy task. The TXP should be just enough, so that every node is able to communicate with the sink either directly or through multiple hops. The *TempLab* testbed has a very high node density. As the goal is to study RPL using multiple hops, a quite small TXP has to be selected. Otherwise all the nodes would connect directly to the sink. Minimizing the hop count is inherent to both ETX and OF0 while the link quality only partly influences the link selection.

To determine a sensible TXP for the testbed, the `connectivity-graph` tool can be used. It has been provided and extended to fit the needs of this thesis. Using this tool, one node at a time sends 100 broadcast messages. All other nodes are recording all messages received and their RSSI. This experiment is repeated using multiple TXPs.



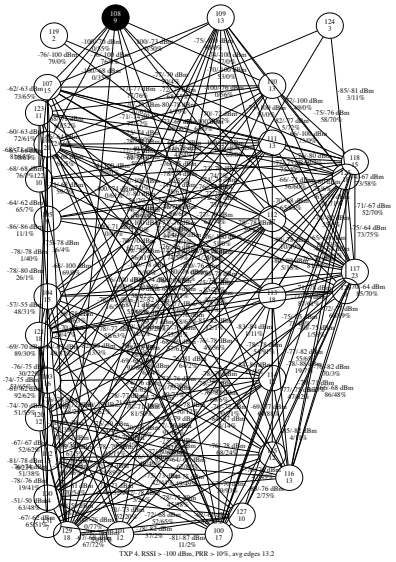
(a) Using TXP 4 (−22 dBm), avg edges 5.2 (b) Using TXP 7 (−15 dBm), avg edges 8.9

Figure 3.2: Testbed links with a PRR higher than 10 % and a RSSI higher than −73 dBm.

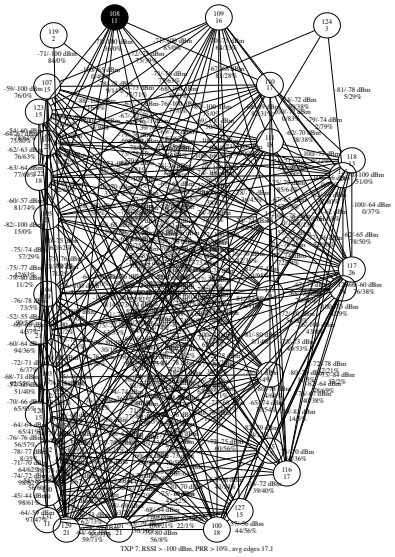
The Figure 3.2 depicts some results of these connectivity tests. The links are filtered to show only “good" connections with a PRR higher than 10 % and a RSSI higher than −73 dBm. The −73 dBm are selected to have some extra margin above the default CCA value of −77 dBm.

Selecting the TXP 4 as shown in Figure 3.2a seems a good fit: almost all nodes are still connected to some other nodes (average 5.2 edges), but there are only 4 direct connection to the sink (Node 108). In comparison, the TXP 3 as shown in Figure 3.3c has almost no

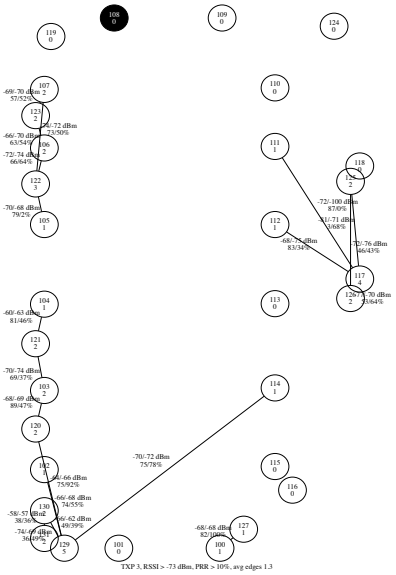
links whereas TXP 7 as shown in Figure 3.2b has a lot more links were more are direct connections to the sink.
Filtering the links by requiring a RSSI above $-73\,\mathrm{dBm}$ helps a lot as can been seen in Figure 3.3. This is more evident when looking at Figure 3.3c and Figure 3.3d. Without filtering there are many links even with TXP 3, but those are only of bad quality. In comparison between TXP 3 and TXP 4, selecting TXP 4 seems like the lowest TXP with enough good links.



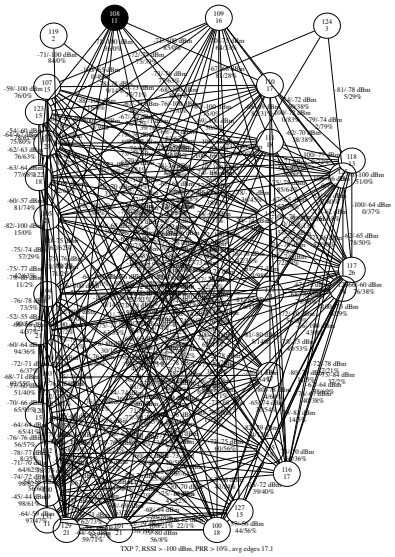
(a) Using TXP 4 ($-22\,\mathrm{dBm}$), RSSI $>-100\,\mathrm{dBm}$



(b) Using TXP 7 ($-15\,\mathrm{dBm}$), RSSI $>-100\,\mathrm{dBm}$



(c) Using TXP 3 ($-25\,\mathrm{dBm}$), RSSI $>-73\,\mathrm{dBm}$

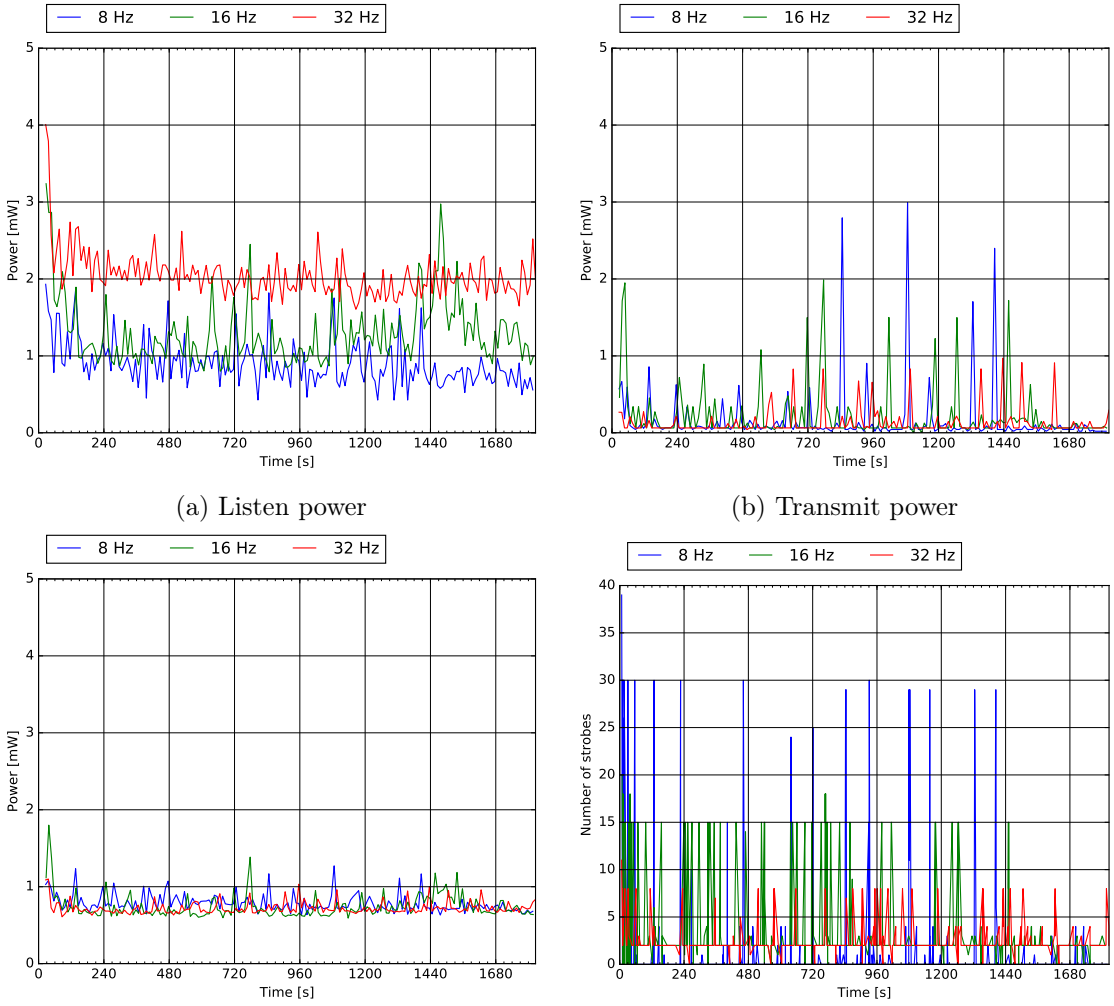


(d) Using TXP 3 ($-25\,\mathrm{dBm}$), RSSI $>-100\,\mathrm{dBm}$

Figure 3.3: Testbed links with a PRR higher than 10 % and various TXPs.

3.2.2 Determining the channel check rate

Figure 3.4 compares the power consumption of an application running `ContikiMAC` with different CCR values (8 Hz, 16 Hz, 32 Hz). After the network startup the power consumption reaches a maximum but after this decreases rapidly, as (i) less control packets are sent and (ii) the phase lock only works after some packets have been received. Using 32 Hz has a higher power consumption, while listening for packets as the radio is awake more often, shown in Figure 3.4a. While sending messages (Figure 3.4b) a higher CCR leads to a lower power consumption as less strobe packets need to be sent in `ContikiMAC` as shown in Figure 3.4d and thus the radio can be turned off faster.
The CPU power consumption is almost identical, as expected. This is the time when the radio is not active and only the CPU is executing code.



(a) Listen power

(b) Transmit power

(c) CPU power

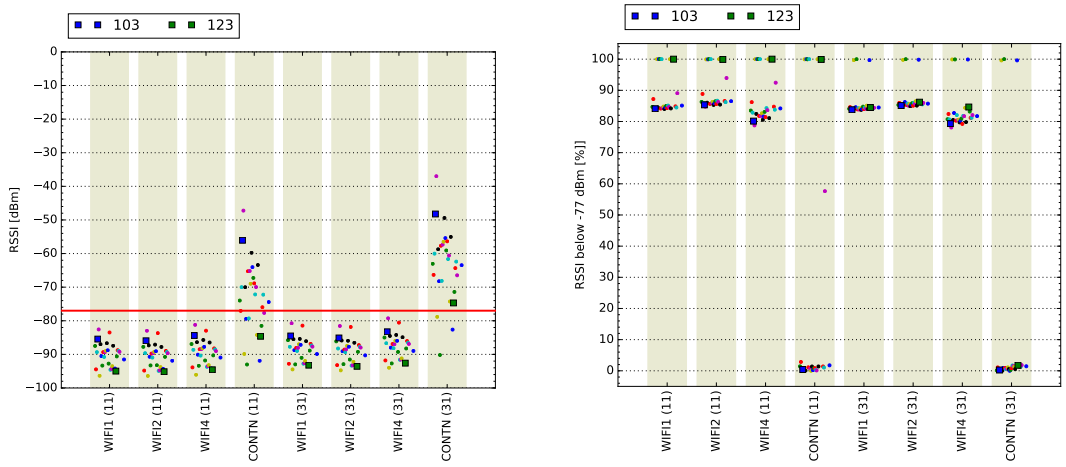
(d) Number of strobe packets

Figure 3.4: Comparing the power usage and number of strobes using different CCR.

The channel check rate used for all other experiments in this thesis is 32 Hz, as less strobe packets need to be transmitted and also the influence of the RDC layer on the routing is reduced.

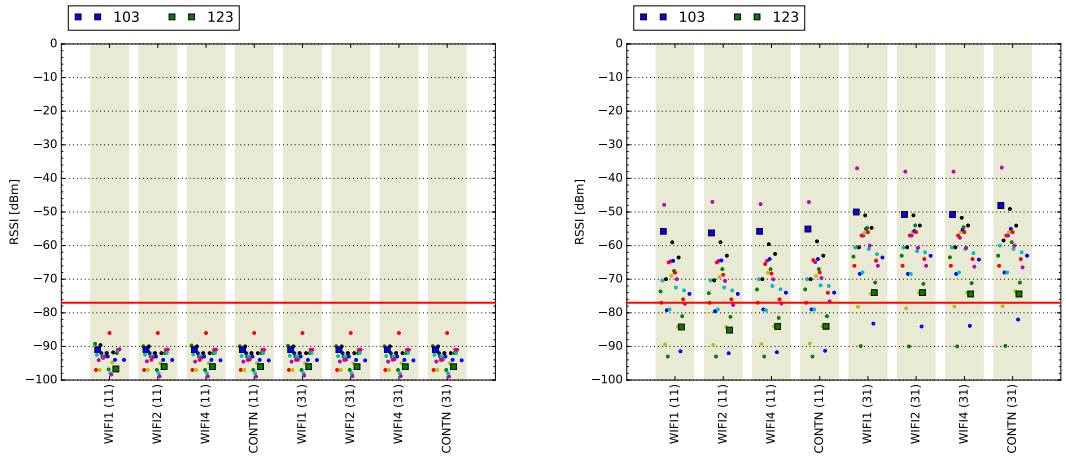
3.2.3 Determine the noise impact

The jammer generates interference at one location in the test bed. Each node is affected differently. This section determines the noise impact at each node depending on the JamLab power level of 11 (−10 dBm) and 31 (0 dBm). Every nodes runs the `noise_analyzer` as described in Section 2.5.2 and records the RSSI at a high sample rate.



(a) Average RSSI while jamming (b) % of RSSI readings below −77 dBm

Figure 3.5: RSSI at various levels of interference.

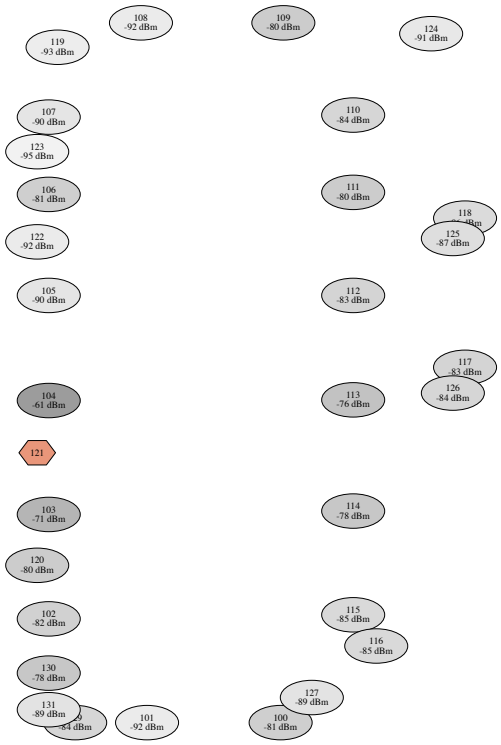


(a) Before jamming (b) During jamming

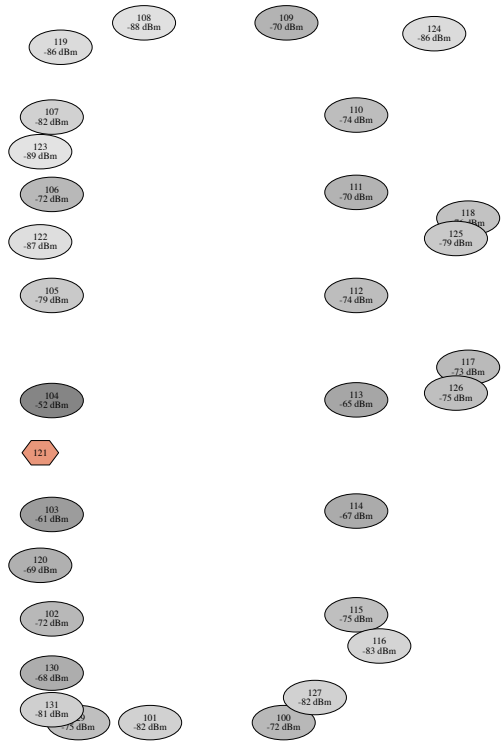
Figure 3.6: Noise floor before and during jamming at various levels of interference.

Figure 3.5a shows the average noise for each node. Each dot represents a single node while the nodes 103 and 123 are highlighted using a larger square. The dots are offset horizontally to better distinguish them and they ordered by the node id. For a successful communication using `ContikiMAC` the RSSI value should be below the CCA threshold. A CCA threshold of −77 dBm is depicted using the red line.

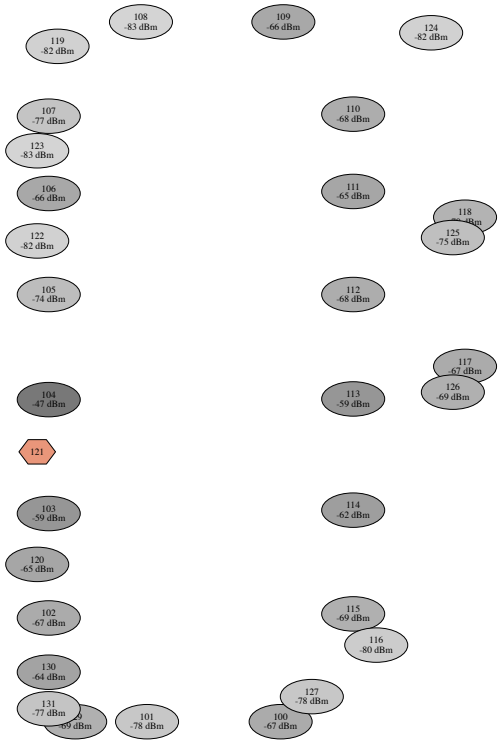
The average number of RSSI readings below −77 dBm for each interference level is shown in Figure 3.5b. For `JL_WIFI1` and `JL_WIFI2` the percentage is about 85 %. For `JL_WIFI4` this value is slightly closer at 80 % indicating the interference type behave indeed differently. For `JL_CONTN` this number drops for most nodes below 5 % making a communication almost impossible. Some nodes like node 123 are not affected by the jammer due to their physical location. Most of the RSSI readings of this node are below −77 dBm, except with the strongest interference level (`JL_CONTN` with TXP 31).
Another way to look at the RSSI readings is the noise floor as shown in Figure 3.6. The noise floor does not use the average instead the maximum of the read RSSI values is used. To be more specific the value where 95 % of the readings lay below is used to filter some extreme outliers.
Without any added noise the noise floor provides the sensitivity of the radio. It should be at around −95 dBm for the CC2420 of the Tmote Sky according to the data sheet [31]. Figure 3.6a shows that the noise floor is indeed at around −95 dBm before turning on the jammer.
Upon turning on the jammer as shown in Figure 3.6a the noise floor increases. The value for each type of interference stays almost the same as long as the TXP is not changed on purpose. This is an expected behaviour and implementation detail of JamLab. The various interference patterns are emulated by switching the transmitter on and off using a probability function, but the TXP is not changed.
The following Figure 3.7 shows four plots depicting the noise impact at each node in the testbed. For these plots the jammer uses `JL_CONTN` and the transmission powers 3, 7, 11 and 31. A darker tone indicates a higher impact (noise floor). It can be observed that the impact not only depends on the nodes physical distance from the jammer, but also on the antenna orientation and on the presence of obstacles in the room.



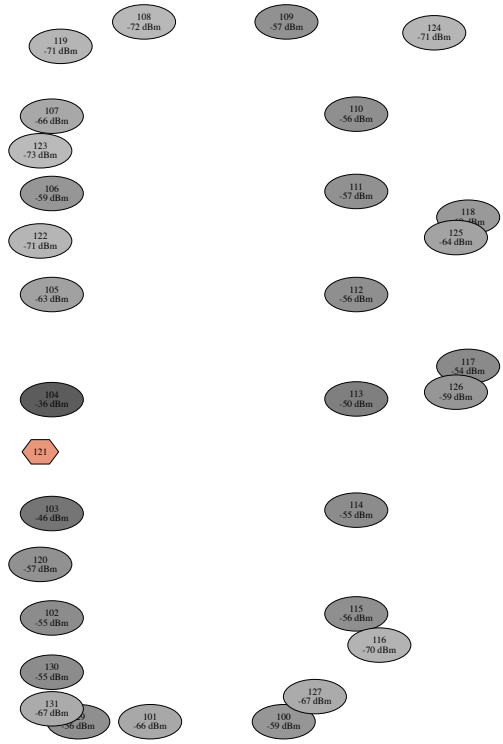
(a) Using TXP 3 (−25 dBm)



(b) Using TXP 7 (−15 dBm)



(c) Using TXP 11 (−10 dBm)

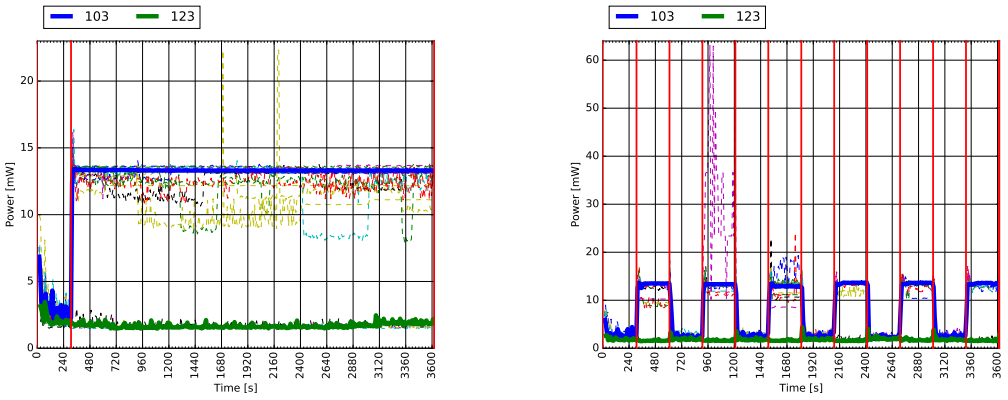


(d) Using TXP 31 (0 dBm)

Figure 3.7: Testbed noise impact at every node using various power levels.

3.2.4 Validate energy consumption

The jammer affects the noise level of the nodes differently depending on their, physical location, as shown in Section 3.2.3. This section takes a look at each nodes energy consumption while subjected to interference. Nodes affected by the jammer should also show a higher energy consumption. The energy consumption while listening for packets of one of the experiments presented in Section 3.3.2 is shown in Figure 3.8.



(a) `JL_CONTN`, permanent (b) `JL_CONTN`, periodic

Figure 3.8: Average listening power for the network with a jammer.

Figure 3.8a shows the power consumption of the network over time. After around 300 s a jammer with a constant carrier is activated (red vertical line). It is kept on for the rest of the experiment. This increases the power usage for most nodes immediately. Each line represent a single node whereas two nodes are highlighted. Node 103 (blue) is affected whereas the node 123 (green) has no increased energy consumption. According to the noise impact from Section 3.2.3 the noise floor at node 123 does not increase and thus it is unaffected by the jammer.
In Figure 3.8b a constant carrier jammer is enabled, but with an alternating on-off period of 5 minutes. The energy also increases while the jammer is on and returns to a lower value as soon as the jammer is turned off. One of the nodes also has keeps its radio on significantly long than the others. This is shown as spikes with a maximum over 60 mW. The maximum power if the radio is on $20.0\,\text{mA} \cdot 3.3\,\text{V} = 66\,\text{mW}$ (See also Section 2.3.2). This verifies the expectations: no increased power for unaffected nodes and an increased power consumption for affected nodes. This also validates that the energy measurements are plausible.

3.2.5 Tools

In order to (i) run multiple experiments with different parameters, (ii) collect the log files and (iii) extract the evaluation metrics described in Section 3.1, some new tools are needed, as the existing ones did not fit this thesis requirements. Therefore, some new scripts and tools were written.
Existing tools such as `collect-view` only operate on data collected at the sink, which is sufficient if there is no other back channel. Using a testbed allows collection of data on every node as each of them is connected to a server.
The new tools consist of some `bash` scripts and a `python` log file parser. The main idea of the log file parser is to use the existing Contiki OS debug output, written to a text file per node. The files are then parsed using a python script that generates python objects and data structures. After parsing all the data multiple tools can operate on it and extract information as needed.
The nodes are configured to print as much debug data as possible, leaving aggregation to the computer running the scripts. This allows for offline data processing and statistic generations as needed. The offline computation of statistics allows to *live watch* an ongoing experiment by parsing the currently written log files as there is no need to wait for the end of an experiment. In fact the length of an experiment is not compiled into the nodes firmware, but runs as long as data is collected.

Log file parsing

The most notable scripts in regard to log file processing are:

- `LogParser.py`
  This python class is the main component of the log file parser implementation. The parser starts by scanning the current directory for text files and fills a list of nodes using the `NodeData` class as a data container. As parsing all the text files is a quite time consuming process the `LogParser` class also manages a cached version of this data. To cache the data, all the python objects are serialized using the `pickle`[3] module. This *pickled* data file can be read about 20 times faster than the original log files as shown in Listing 3.1. If a cache file (`parsed_logs.pickle`) exists, it is used automatically as long as the script does not force new parse process.
  After the data of all nodes is read, all the timestamps are converted to a relative time based on the startup of the sink.

```
Parsing done in 89.32 seconds for 519.61 MB (5.82 MB/s) of log files.
    ↪ Cached in ./parsed_logs.pickle (83.88 MB).

Loaded cached log objects in 4.44 seconds for 519.61 MB (117.05 MB/s) of
    ↪  log files. Cache file size 83.88 MB (18.90 MB/s).
```

Listing 3.1: Comparison of parsing speed and speed of loading cached data.

- `NodeData.py`
  The `NodeData` class holds all the parsed data for a single node. The data is parsed

[3] https://docs.python.org/3.5/library/pickle.html

line by line. For each line some characters at the beginning (prefix) are compared to determine the software module responsible to parse the rest of it. This splits the logic into multiple classes where each of them holds the respective data. Currently, these classes are implemented: `ContikiMac`, `EnergestStats`, `JamLab`, `NoiseAnalyzer`, `RIMEStats`, `RPLStats`, `SensorData` and `UIPStats`. Some more general data is also parsed and stored in the `NodeData` class directly.
The parsing is typically performed using a regular expression for each prefix.

- `Statistics.py`
  The `Statistics` class calculates some statistics across all the nodes after parsing the data. Most metrics defined in Section 3.1 are computed here. This data is also part of the pickled cache file.

- `parse-node-addr.py`
  This script parses the collected data and prints a list of the mapping between the node id and the last two byte of the nodes address. This address is used when sending messages to other nodes and is also displayed in `collect-view`. The mapping to a node id is an important utility as the node id is written on every node, the log files use the node id as its filename and also the map of the testbed uses the node id to identify each node. As sample output is shown in Listing 3.2.

```
100 ->   49.43 Parents: 229.69 (360x),
101 ->  254.37 Parents: 177.37 (349x), 169.27 (11x),
102 -> 250.238 Parents: 49.43 (362x),
103 -> 143.115 Parents: 49.43 (361x),
104 ->  127.62 Parents: 229.69 (361x),
105 ->    59.0 Parents: 120.250 (275x), 127.104 (54x), 216.66 (6x),
106 ->  216.66 Parents: 229.69 (361x),
107 -> 120.250 Parents: 229.69 (362x),
108 ->  229.69 Parents: SINK !
109 ->  169.27 Parents: 33.16 (339x), 133.75 (20x),

...
```

Listing 3.2: Sample output of the `parse-node-addr.py` script.

- `generate-plot-ccr.py`
  This script creates the four plots comparing different CCR values as shown in Figure 3.4. To run the script multiple experiment log folders need to be passed as command line arguments. The data of each folder is then combined into the plots. The plots are created using the `matplotlib`[4] and written to a image file.

- `generate-plot-noise-impact.py`
  This script creates the four plots showing the noise impact as used in Section 3.2.3. It requires at least one folder as command line argument and combines the given folders to generate the plots.

- `generate-plot-energy.py`
  This script creates plots showing the power usage over time similar to the previous

[4] http://matplotlib.org

script. Three plots are created (listen, TX, CPU) and saved to the current working folder. Examples of this script are shown in Section 3.2.4.

- `generate-node-map.py`
  This tool creates a series of plots to visualize the RPL graph and its parent selection over time with the real nodes position. The script also generates a minimalistic HTML user interface to browse between the timestamps as shown in Figure 3.9. The plots are created using `GraphViz`[5].

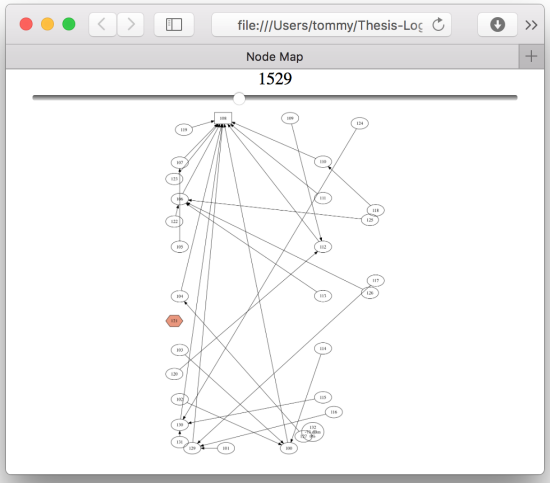


Figure 3.9: Node map visualized using `dot` and HTML.

- `generate-parent-graph.py`
  This is another tool to get an overview of the network and the connectivity between nodes. One example plot is shown in Figure 3.10. The labels on the edges indicate the PRR and the ETX, less used edges are plotted using lighter colors.

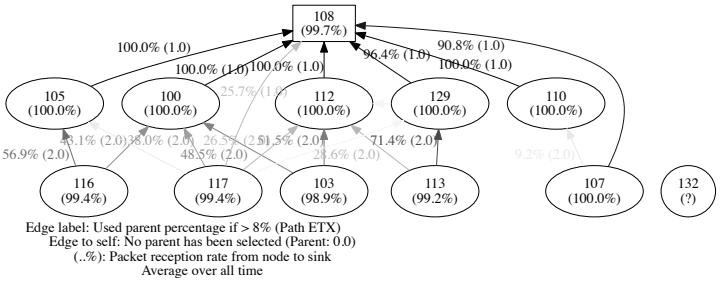


Figure 3.10: Example output of `generate-parent-graph.py`.

[5] http://www.graphviz.org

- `generate-time-graph.py`
  This tool generates a huge plot including every node and its packet transmission status of an experiment. It clearly shows which nodes are effected by JamLab and gives a general visual overview of the network. The plot is also created using the `matplotlib` python package. Every event creates a new dot in the graph, the most important events of the application layer (bigger dots) are: packet sent (red), packet received at sink (turquoise), parent changed (blue). Printed as smaller dots are events from the MAC layer. The small red dots indicate a collision while sending a packet, the yellow dots show that the channel was busy and no attempt to sent packet was made.
  More details of the results obtained using this tool are presented in Section 3.3.

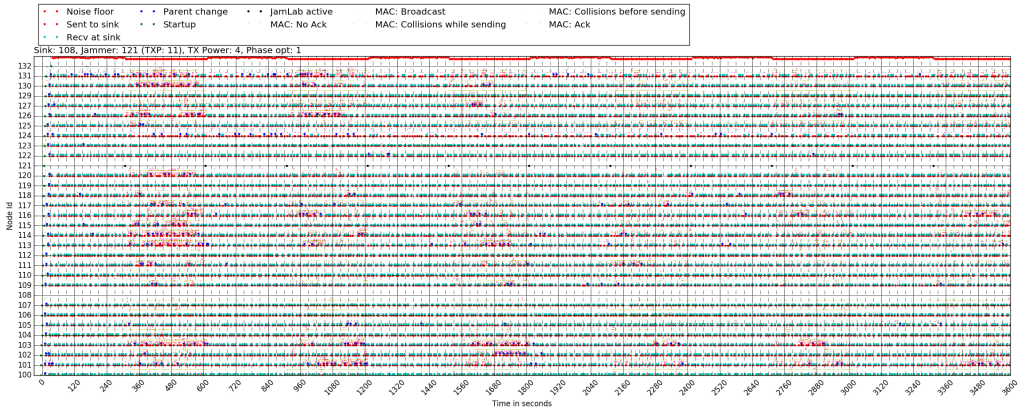


Figure 3.11: Example time graph as generated by `generate-time-graph.py`.

- `generate-traceroute.py`
  This tool uses the hop-by-hop trace route functionality as described in Section 2.5.3. It extracts the longest path of each packet sent and writes this information to a text file.
  A sample output for node 131 is shown in Listing 3.3. It includes the timestamp when the packet was sent, its sequence id and then every node that received the packet including the RSSI with which it was received. If a packet did not reach the `SINK`, it is marked as `LOST`.

```
Trace for node 131
 32.33 |    0: 131 -> 127 (-77) -> 104 (-74) -> 108 (-86) = SINK
 42.05 |    1: 131 -> 127 (-77) = LOST
 45.57 |    2: 131 -> 127 (-79) -> 100 (-61) -> 108 (-80) = SINK
 58.25 |    3: 131 -> 127 (-77) -> 100 (-62) -> 108 (-80) = SINK
 70.26 |    4: 131 -> 127 (-77) -> 100 (-61) -> 108 (-80) = SINK
 79.20 |    5: 131 -> 127 (-79) -> 100 (-61) -> 108 (-80) = SINK
 86.59 |    6: 131 -> 127 (-78) -> 100 (-61) -> 108 (-80) = SINK
 97.12 |    7: 131 = LOST
108.85 |    8: 131 -> 129 (-66) -> 108 (-68) = SINK

...
```

Listing 3.3: Sample trace output using the `generate-traceroute.py` script.

- `generate-statistics.py`
  This tool uses the data from the `Statistics` class to generates a nice table overview as shown in Figure 3.12. This script only contains logic to visualize the data: all the computation happens already at *parse-time* so that other tools can still easily access this data.

stats.log — Open with Console

Node ID	Start	Recv packets	Sent packets	PRR %	ICMP sent	Parent Changes	RSSI dBm	E RX	E TX	E CPU	E LPM
100	0	360	360	100.0	94	1	−72	5.38	0.11	0.78	0.18
103	−2	335	361	92.8	137	15	−73	5.97	0.32	1.15	0.18
105	1	357	359	99.4	96	4	−76	2.26	0.23	0.71	0.18
107	0	360	360	100.0	87	1	−68	3.03	0.11	0.84	0.18
S 108	0	0	0	0.0	5	0	−75	0.00	0.00	0.00	0.00
110	1	360	361	99.7	49	2	−79	2.86	0.07	0.60	0.18
112	2	361	361	100.0	99	1	−69	5.87	0.12	0.95	0.18
113	3	250	356	70.2	184	25	−74	6.12	0.46	1.37	0.18
116	4	349	359	97.2	98	6	−72	5.69	0.23	0.94	0.18
117	5	360	360	100.0	84	5	−77	5.07	0.21	0.84	0.18
J 121	7	0	0	0.0	0	0	0	0.00	0.00	0.00	0.00
129	12	359	359	100.0	75	2	−75	5.23	0.08	0.75	0.18
N 132	14	0	0	0.0	0	0	0	0.00	0.00	0.00	0.00
Sum/Avg		3451	3596	96.0	1008	62	−74	4.75	0.19	0.89	0.18

Figure 3.12: Sample statistics generated using `generate-statistics.py`.

- `aggregate-stats.py`
  This script expects multiple experiment directories as its launch arguments and aggregates some statistics into a single comma-separated values (CSV) file as shown in Figure 3.13. This is also creates a `TEX` file containing a formatted table as used in Table 3.1.

aggregated-stats_com_no_noise.csv — Open with Microsoft Excel

run id	RDC	CCR [Hz]	CCA [dBm]	Jam Type	Jam Power	PRR [%%]	RSSI [dBm]	P RX [mW]	P TX [mW]	P CPU [mW]	P LPM [mW]
log_2016-08-10_23-46-41_69_rpl_txp4_cca77_ccr8_cm_etx_noise_132	ContikiMAC	8	-77			94	-73	0.89	0.30	0.80	0.18
log_2016-08-11_00-17-39_51_rpl_txp4_cca77_ccr32_cm_etx_noise_132	ContikiMAC	32	-77			94	-73	2.30	0.16	0.81	0.18
log_2016-08-11_00-48-57_62_rpl_txp4_cca77_ccr32_cm_of0_noise_132	ContikiMAC	32	-77			80	-74	2.35	0.21	0.78	0.18
log_2016-08-11_01-19-45_67_rpl_txp4_cca77_ccr32_cm_etx_nullrdc_noise_132	nullrdc	128	-77			99	0	65.96	0.03	0.86	0.18
log_2016-08-11_01-50-47_68_rpl_txp4_cca77_ccr32_cm_of0_nullrdc_noise_132	nullrdc	128	-77			99	0	65.96	0.04	0.88	0.18

Figure 3.13: Aggregate statistics collected from multiple experiments.

- `aggregate-stats-noise.py`
  This scripts combines noise floor measurements into a CSV file. It is similar to the `generate-plot-noise-impact.py` tool, but instead of plot the output is the raw data to be used in other software.

Testbed and experiment management

To better manage experiments on the TempLab testbed the following scripts where used, extended or newly created:

- `check_bot.sh`
  The testbed is shared among multiple users, the usage is lightly coupled to a shared calender, but it is not enforced. As the testbed might also be available earlier or later, a tool to check the testbed status has been developed. It checks every minute (using `cron`) if the testbed is currently free and sends a notification to a Telegram[6] group chat[7] as shown in Figure 3.14. These messages are also received as push notifications on a smart phone. Sending messages to a telegram user is quite easy.
  One needs to register a Telegram bot using the BotFather[8] to obtain a `token`, which is then used to make calls to an application programming interface (API). Using the `token`, sending a message to the group requires only one HTTP request.
  The same functionality is also used to send a private message if a series of experiments finishes either due to an error or if no more experiments are scheduled.

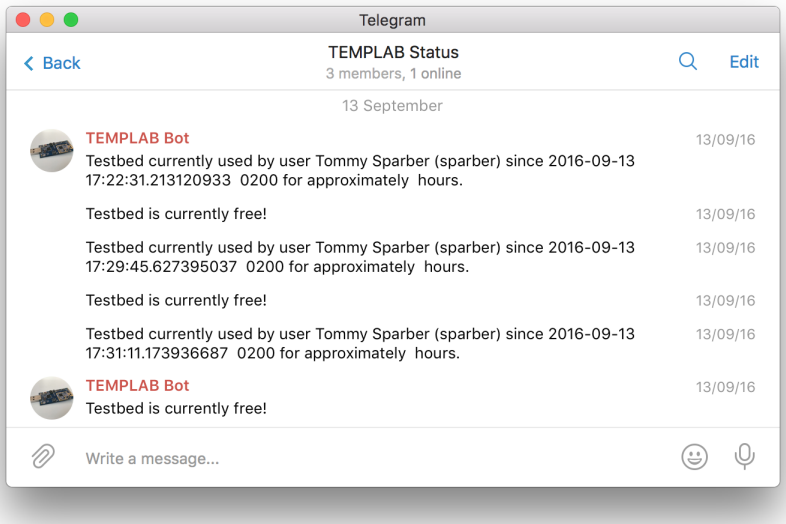


Figure 3.14: TempLab Status updates sent to a Telegram group chat.

- `iti-list-remaining-nodes.py`
  The testbed also includes nodes that may not be used as part of the experiment. As it is not possible to disable them without physically disconnecting them, a known firmware needs to be programmed to ensure that no unwanted firmware interferes with the experiment. The current experiment definition needs to list any node and its firmware. As the testbed and especially the not needed nodes change often, this

[6] https://telegram.org
[7] https://telegram.me/joinchat/B6D-3wjj0hJb699ETN8now
[8] https://core.telegram.org/bots#6-botfather

script has been created to list all available nodes while also filtering all nodes passed as arguments as shown in Listing 3.4.

```
$ NODES_TO_BE_PROGRAMMED_1=( N00 N01 N03 N04 N05 N06 N07 N08 N09 N10 )
$ iti-list-remaining-nodes.py ${NODES_TO_BE_PROGRAMMED_1[@]}
N02 N28 N29 N30 N31 N32 N33 N34 N35 N36 N37 N38 N39 N44 N45 N46 N47 N53
```

Listing 3.4: Sample output of the `iti-list-remaining-nodes.py` script.

- `iti-program_testbed`
  This script programs the whole testbed using a given configuration. It has been extended to be better configurable. It programs every given node in parallel including multiple retries. The programming of some nodes may fail, thus including multiple retries helps to make the experiment run more stable. If no configuration file is given, a default simple example is uploaded. The configuration file needs to define a `program_firmware_list` function that may include multiple calls to the provided `program_firmware` function. An example configuration file is shown in Listing 3.5.

```
# Testbed nodes to be programmed and with which firmware
NODES_TO_BE_PROGRAMMED=( N00 N01 N02 N03 N04 N05 N06 N07 N08 N09 N10 )
NODES_INACTIVE=( $(iti-list-remaining-nodes.py ${NODES_TO_BE_PROGRAMMED[
    ↪ @]}) )

FIRMWARE_FILE_1="simple_example.ihex"
FIRMWARE_LEDS_BLINK="tinyos_blink_firmware/main.ihex"

function program_firmware_list ()
{
    program_firmware "${FIRMWARE_LEDS_BLINK}" NODES_INACTIVE[@]
    program_firmware "${FIRMWARE_FILE_1}" NODES_TO_BE_PROGRAMMED[@]
}
```

Listing 3.5: Example configuration file.

- `run-iti-prog-listen-close.sh`
  This script uses the `iti-program_testbed` and the `traces_start_tty.sh` script to program the the testbed, listen for log messages for a specified time and then stops the logging and returns to the caller. It also creates new folders based on the experiment configuration name.

3.3 Experiments using RPL running on Contiki OS

This section contains results of the experimental campaign.
The experiments started on a larger scale using about 30 nodes and were then reduced to a smaller set of about 12 nodes. For the following experiments the CCR was chosen to be 32 Hz according to Section 3.2.2 and the transmission power was set to 4 (−22 dBm) according to Section 3.2.1.

3.3.1 Performance of RPL with ETX & OF0 without jamming

The first investigated RPL setup uses Contiki's default settings: `ContikiMAC` as RDC and a CCA level of −77 dBm. This setup is compared to a configuration without any RDC (`nullrdc`). In the case of `nullrdc` the radio stays always on and the CCA influences only the collision detection while sending a packet. In the case of `ContikiMAC` the CCA check is also used to decide if the radio should be kept on to possibly receive a packet. The objective functions used are ETX and OF0.
The goal of the experiments in this section is to show how Contiki OS's RPL performs in a test setup without any interference. The sink used is the same as in the previous sections (Node 108).

RDC	Nodes	Objective Function	PRR [%]	Power RX [mW]	Power TX [mW]	Power CPU [mW]	Power LPM [mW]	Parent changes/node	Parent changes/packet	Hops	Nodes PRR >90%
ContikiMAC	30	ETX	94	2.30	0.16	0.81	0.18	4.000	0.022	1.80	27
ContikiMAC	30	OF0	80	2.35	0.21	0.78	0.18	1.069	0.006	1.86	22
nullrdc	30	ETX	100	65.96	0.03	0.86	0.18	1.621	0.009	1.27	29
nullrdc	30	OF0	100	65.96	0.04	0.88	0.18	1.103	0.006	1.59	29
ContikiMAC	11	ETX	100	1.73	0.08	0.60	0.18	2.000	0.006	1.39	10
ContikiMAC	11	OF0	85	1.90	0.18	0.62	0.18	1.000	0.003	1.70	8

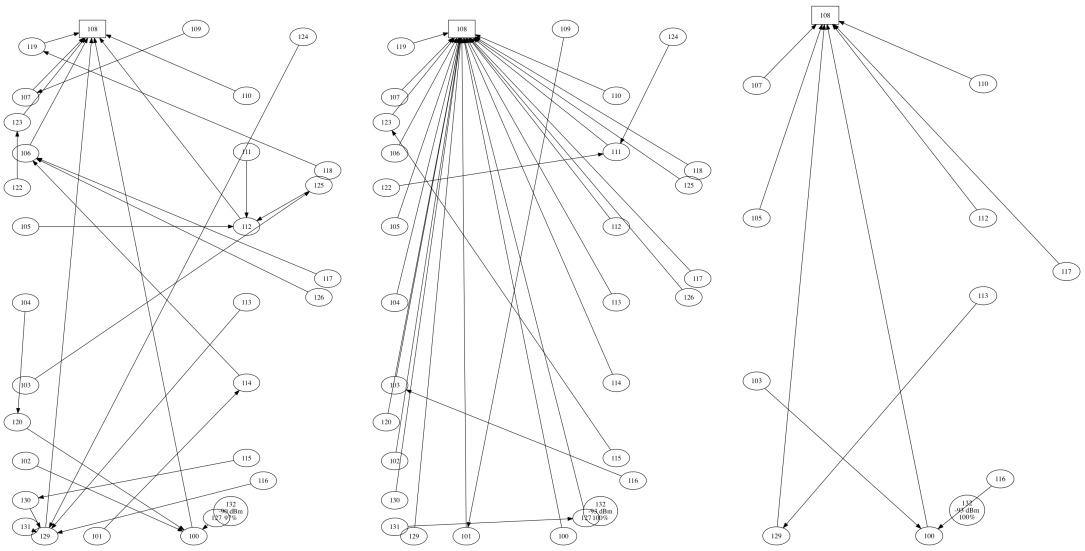
Table 3.1: Comparing `ContikiMAC` with `nullrdc` using RPL ETX and OF0 OF.

Packet reception rate. Table 3.1 shows the results obtained after running six different experiment configurations. Using ETX and `ContikiMAC` seems to work quite well. In comparison with OF0, ETX performs better. Using ETX and `ContikiMAC` almost all nodes (27 out of 29 sending nodes) manage to connect to the sink with a PRR higher than 90 % where as using OF0 only 22 nodes result in a stable connection. This also limits the average PRR to 80 % in comparison to 94 % of ETX.
Both objective functions achieve a PRR of 100 % when using `nullrdc`. This is the first indication that the RDC layer can have an influence on the performance of RPL. The channel check rate used in `ContikiMAC` is 32 Hz, which is already 4 times higher than the default 8 Hz. Another advantage of `nullrdc` is that every packet the receiver would be able

to decode (i.e., has an RSSI higher than about −90 dBm) is received and processed. In `ContikiMAC` the RSSI has to be above the configured CCA level of −77 dBm to be reliable received.

Power consumption. ETX performs slightly better than OF0 with respect to power consumption. A lost packet has the highest power consumption, as the experiments using OF0 have a lower PRR also a slightly higher power consumption is expected. The power consumption of `nullrdc` is of course a lot higher as the radio is always kept on. The power consumption of the radio while listening for packets is 65.96 mW, which close to the maximum of $20.0\,\text{mA} \cdot 3.3\,\text{V} = 66\,\text{mW}$ (according to Equation 2.6). The missing part is measured while the radio is transmitting packets.

This series of experiments also includes two smaller test runs on a sparse network consisting of 11 nodes distributed across the room. The goal is to compare ETX and OF0 on less nodes which reduces the congestion on the channel as less nodes are competing for medium access at the same time. In this setup ETX works also better and achieves a PRR of 100 %. Using OF0 two nodes did not find a good connection and had a PRR below 90 %. The experiments on the sparse network were only run using `ContikiMAC`. The experiments using `nullrdc` are only performed to get a baseline for the other experiments. `nullrdc` is not a suitable RDC as the energy consumption is a lot higher which reduces the expected runtime on a battery significantly.



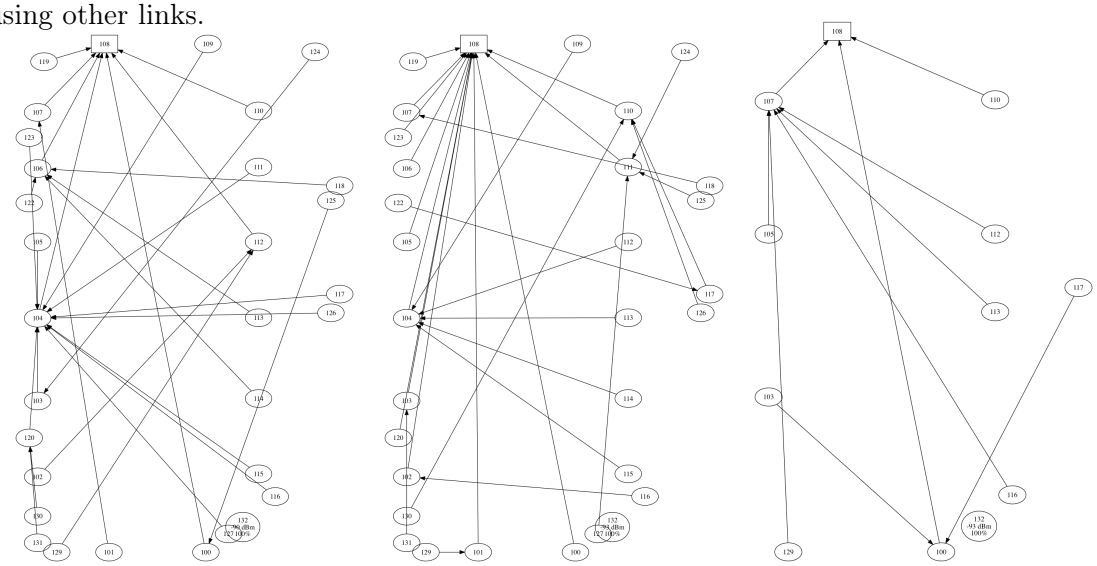
(a) `ContikiMAC` (b) `nullrdc` (c) `ContikiMAC`, sparse

Figure 3.15: Topology of the network while using different RDC protocols and ETX.

Topology. The topology of the networks, as shown in Figure 3.15 for ETX and in Figure 3.16 for OF0, is quite different between `ContikiMAC` and `nullrdc`. With `nullrdc` more nodes are directly connected to the sink (top) as evident in Figure 3.15b. The RSSI on these links is lower as the distance is higher. These links will be easily affected by and fail due to interference, as will be shown in the next sections.

These figures also give another hint, that the RDC has a significant influence on the RPL tree construction and also on its stability and expected resistance to radio interference.

Both objective functions try to minimize the number of hops and the nodes are not allowed to choose a parent whose rank is higher than its own to avoid loops without a global repair. This means, the nodes prefer lower ranked nodes instead of selecting more reliable ones if they received any packets from these nodes. As soon as they received packets and ACKs directly from the sink they also stay connected to it even though the RSSI is lower than using other links.



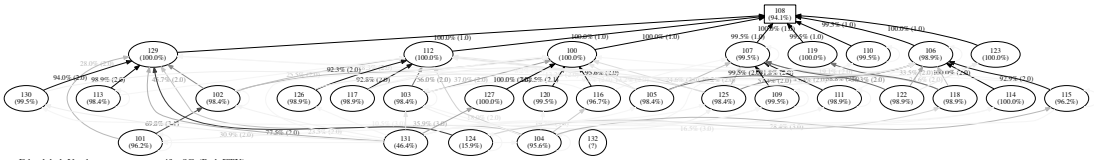
(a) `ContikiMAC` (b) `nullrdc` (c) `ContikiMAC`, sparse

Figure 3.16: Topology of the network while using different RDC protocols and OF0.

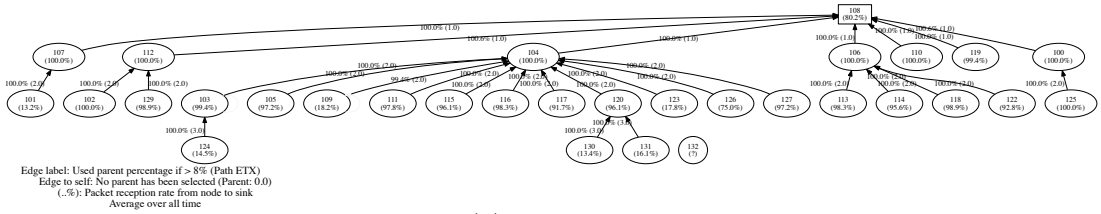
The formed networks while using ETX and OF0 look quite similar. The following plots in Figures 3.17, 3.18 and 3.19 depict the parent selection summarized over an experiment run. Less frequent links are drawn using a lighter colour as they are not so significant. It can be observed, that graphs using OF0 such as Figure 3.17b and Figure 3.18b have less edges as the parent is not often changed.
The number in the nodes circle indicates the PRR whereas the number on the edge between two nodes indicates the percentage that this links was chosen.
Using these plots it is possible to observe, that the number of hops and the stability of the network is quite different between `ContikiMAC` and `nullrdc`. In the `nullrdc` experiment many nodes are directly connected to the sink or at a maximum of two hops. This is reasonable as using `nullrdc` a packet with a lower RSSI is still receivable as the limit is only the sensitivity of the radio and not the CCA level as in `ContikiMAC`.

Lessons learned:

- *RPL is scalable: In a large network of about 30 nodes works quite well.*
- *ETX performs better than OF0.*
- *`nullrdc` as a baseline shows that a PRR of* 100 % *is achievable.*
- *The used RDC (`ContikiMAC`) has an influence on network performance. It requires a certain RSSI to wake up and receive a packet. Due to this some packets are lost.*

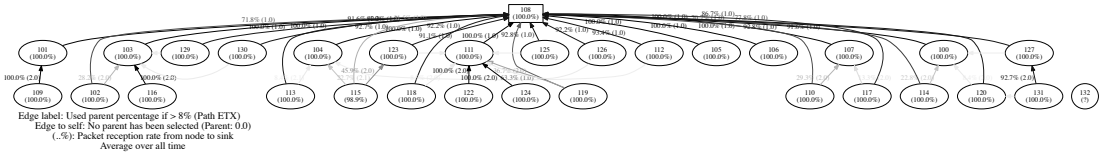


(a) using ETX

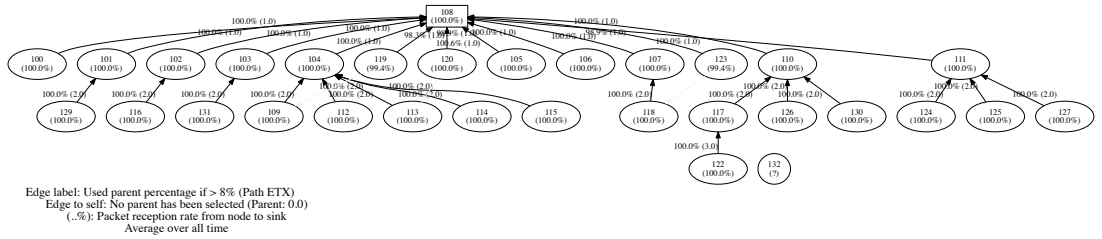


(b) using OF0

Figure 3.17: RPL tree while using `ContikiMAC` on 30 nodes.

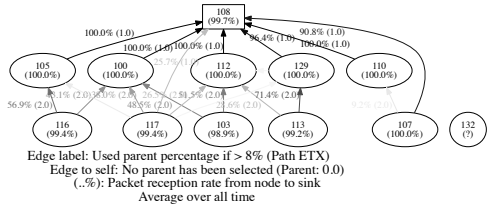


(a) using ETX

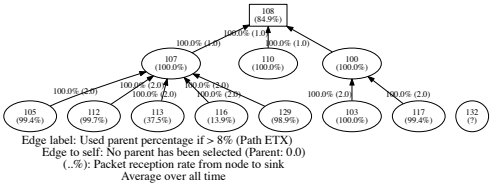


(b) using OF0

Figure 3.18: RPL tree while using `nullrdc` on 30 nodes.



(a) using ETX



(b) using OF0

Figure 3.19: RPL tree while using `ContikiMAC` on 11 nodes.

3.3.2 Performance of RPL with ETX & OF0 and permanent jamming

This section shows the performance of RPL in a network subjected to radio interference. The different interference types are generated using JamLab as described in Section 2.5.1. The experiments are run using `ContikiMAC` and a CCA of −77 dBm. The nodes use transmission power level 4, whereas the jammer uses the power level 11.

Jam Type	Jam Power	Nodes	Objective Function	PRR [%]	RSSI [dBm]	Power RX [mW]	Power TX [mW]	Power CPU [mW]	Power LPM [mW]	Parent changes/node	Nodes PRR >90%
JL_WIFI1	11	30	ETX	97	-74	4.06	0.18	0.89	0.18	3.724	27
JL_WIFI2	11	30	ETX	97	-73	4.22	0.22	0.96	0.18	5.414	28
JL_WIFI4	11	30	ETX	86	-72	6.24	0.33	1.41	0.18	13.241	18
JL_WIFI4	11	30	OF0	67	-73	6.17	0.38	1.39	0.18	0.034	10
JL_CONTN	11	30	ETX	9	-69	10.98	0.04	3.32	0.18	0.517	2
JL_CONTN	11	30	OF0	7	-70	10.65	0.04	2.68	0.18	0.000	2
JL_WIFI4	11	11	ETX	96	-73	5.00	0.20	0.91	0.18	5.100	9
JL_WIFI4	11	11	OF0	75	-74	5.12	0.27	0.99	0.18	0.000	7

Table 3.2: Comparing various interference intensities while using RPL ETX and OF0 OF.

Packet reception rate. Table 3.2 depicts the results of these experiments. The lighter interference types such as `JL_WIFI1` and `JL_WIFI2` have a small impact on network performance. The PRR is even a bit higher at about 97 % compared to 94 % of the previous experiments without noise. This may be just in the range of variations between multiple test run. Most nodes at least have a PRR above 90 %.

Power consumption. In comparison to the experiments from Table 3.1 the average RX power consumption has almost doubled from 2.30 mW to about 4.10 mW. This might be due to more false wake ups from the RDC layer[9] as well as more required transmission attempts while waiting for a clear channel. The average TX, CPU and LPM power consumption stays almost the same, but also increases as the interference level increases. There is also a clear correlation between the interference intensity and the overall power consumption. A node subjected to a higher interference level uses more energy and thus its battery life will be shortened significantly reducing the overall network reliability.

A higher interference level (`JL_WIFI4`) has visibly a higher impact on the PRR and energy consumption. This impact is even more evident when using OF0 instead of ETX. Using OF0, the PRR drops from 86 % to 67 %.

Increasing the burden on the radio channel even more by switching to the continuous carrier jamming mode (`JL_CONTN`), the network completely breaks apart, lowering the PRR

[9]The current experiment data does not measure the false wakeups.

down to 9 %, with only two nodes being able to communicate reliably with the sink. This increases the average RX power consumption even more while only wasting energy without any remarkable communication success. This higher energy consumption is assumed to be due to long false radio wake ups. The TX energy is reduced as almost no packet is sent due to the channel being considered almost always busy.
The sparse network using only 11 nodes is only investigated using `JL_WIFI4` as the lower interference types already have only little impact on the bigger network. The smaller network performs slightly better than the bigger network as shown before and has a higher PRR even under interference. The energy consumption though is still about three times higher than without interference.
Parent changes. An interesting thing to note is also the significantly increased number of parent changes per node. This means the network is less stable, but also that it tries to find new routes to still reach the sink.
Collisions. Generating the time-graph as shown in Figure 3.21 is really helpful to get a better understanding of what is happening to the network.
In both plots node 108 is the sink, 121 is the jammer and 132 records the noise floor at its position.
The jammer is programmed to start at about 300 s (5 min) after the network start up. The impact is clearly visible as the small dots start to display collisions while sending (small red dots), collisions before sending (yellow small dots) as well as not received packets at the sink (missing cyan dots). The plot in Figure 3.21a displays the huge impact of the `JL_WIFI4` jammer whereas the Figure 3.21 depicts the almost completely broken network after turning on the continuous jammer (`JL_CONTN`). Each yellow dot indicates a failed clear channel assessment. With the continuous jammer the noise floor is increased so that the RSSI is almost always above the CCA threshold and thus the channel was assumed busy when an attempt to send a message was made.
This gives a hint, that the clear channel assessment should be investigated further.

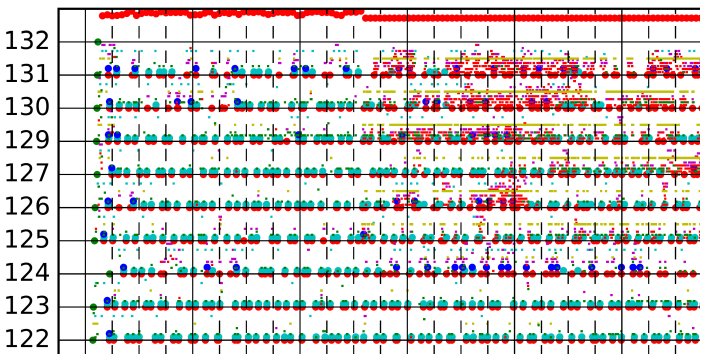


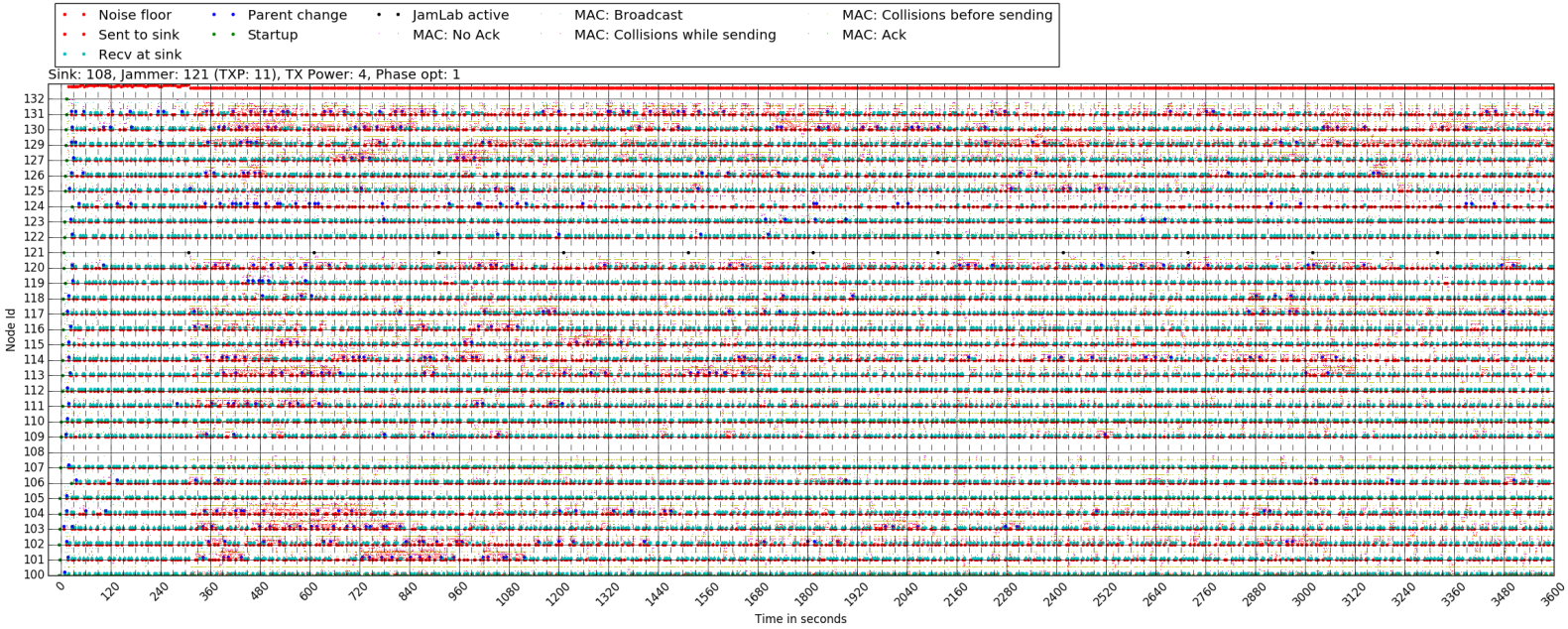
Figure 3.20: Zoomed part out of Figure 3.21a.

Figure 3.20 depicts a zoomed part of the bigger time graph to better illustrate its contents. At the moment when the jammer starts being operational, the noise floor is increased and the nodes start to have problems communicating with each other. Packets are lost or not even sent as the channel is considered busy. As it can been seen, node 124 also starts to rapidly change the preferred parent. Other nodes such as 123 and 122 are not really

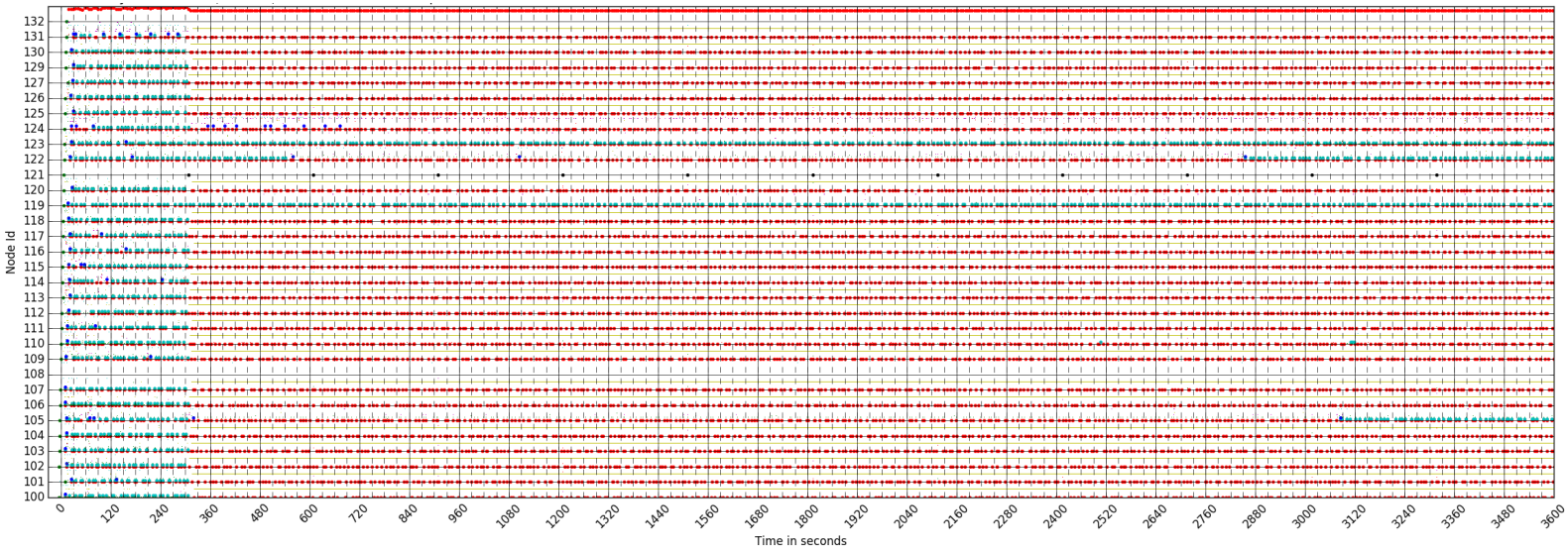
affected by the jammer and are able to communicate regardless of the introduced interference. The jammer affects nodes differently due to there physical location and distance from the jammer. A more in depth look at the noise impact at each node has already been discussed in Section 3.2.3.
Figure 3.22a shows the impact of `JL_WIFI1` on the network. The enabled jammer is only lightly visible, the network is almost unaffected. In Figure 3.22b the network using OF0 is shown while subjected to `JL_WIFI4`. The network clearly has problems in the communication, but the situation does not improve as the OF does not find new, better paths.
The time-graph of `JL_WIFI2` on ETX and `JL_CONTN` of OF0 are not included as they look almost the same as Figure 3.22a and Figure 3.21. The time-graphs of the experiments without noise almost always contain a straight cyan line of successfully received packets and only some blue dots indicating some parent changes.

Lessons learned:

- *ETX performs better (higher PRR) than OF0 even under interference.*
- *As long as the interference is not too strong (`JL_WIFI4`), the network sustains a high PRR.*
- *If the interference is strong (`JL_CONTN`), the network completely breaks and the PRR falls below* 10 %.
- *Interference increases the energy consumption of every affected node,*
- *Interference has a strong influence on the CCA operation of `ContikiMAC`.*
- *A sufficiently strong interferer blocks every attempt to send a message as the CCA check always sees a busy channel.*
- *On the receiving side, interference keeps the radio on longer as `ContikiMAC` assumes there might be a packet. This increases the energy consumption significantly (2-4 times on average in these experiments). This drains their battery faster, making the network less reliable.*
- *Automatically changing the CCA level may help to send messages and to reduce the energy consumption in the presence of interference. It should be investigated further.*

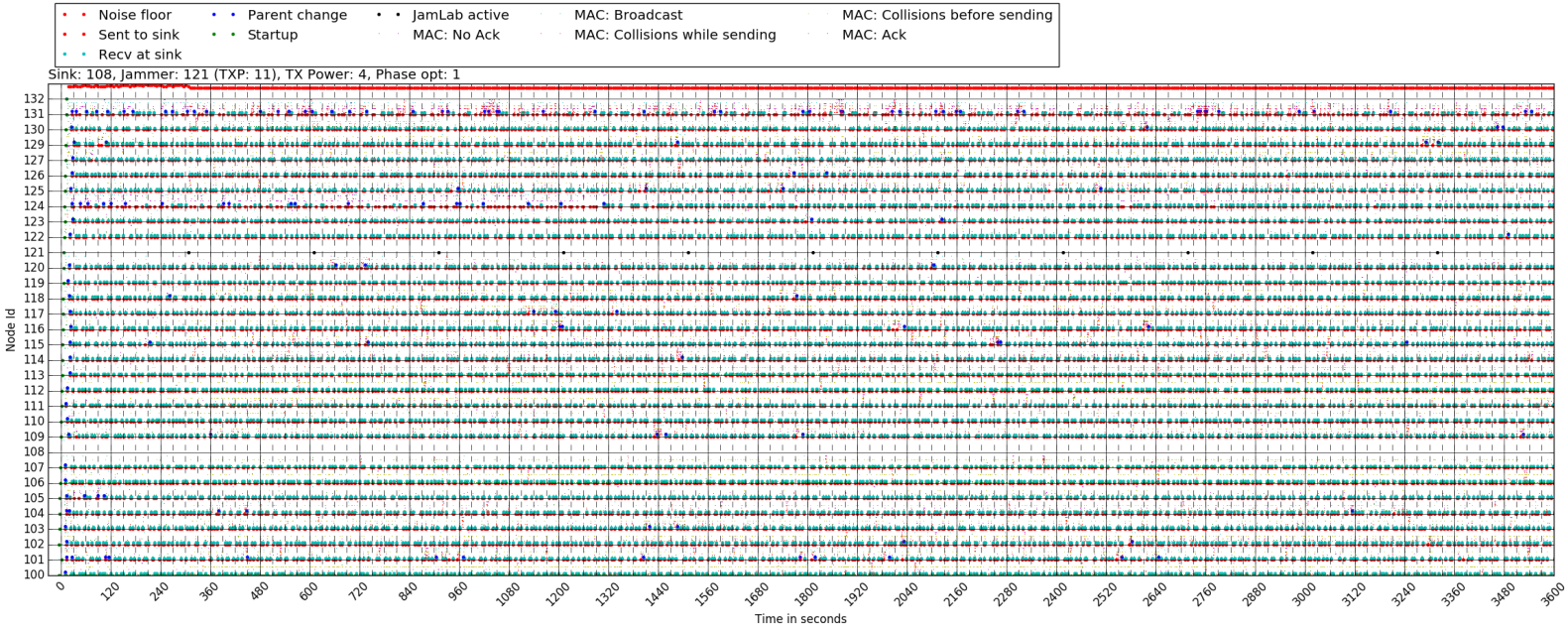


(a) The network over time with a `JL_WIFI4` jammer.

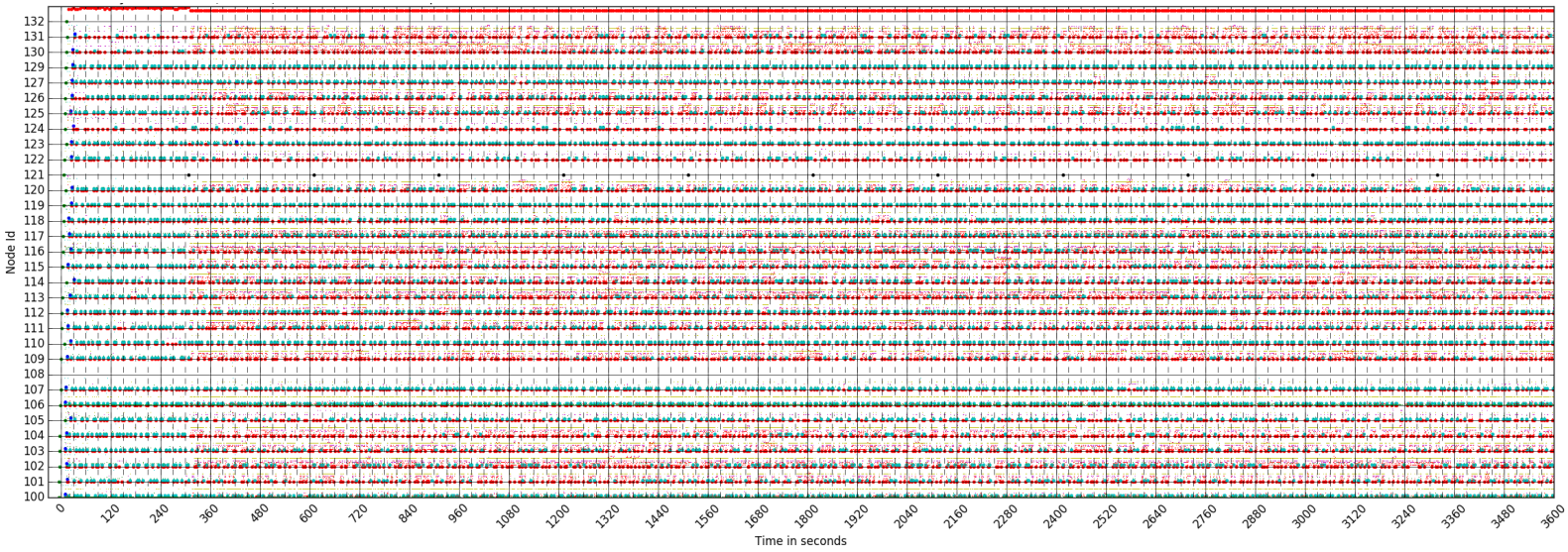


(b) The network over time with a `JL_CONTN` jammer.

Figure 3.21: Collected events of the network from each node over the whole experiment time (ETX).



(a) The network over time with a `JL_WIFI1` jammer and ETX.

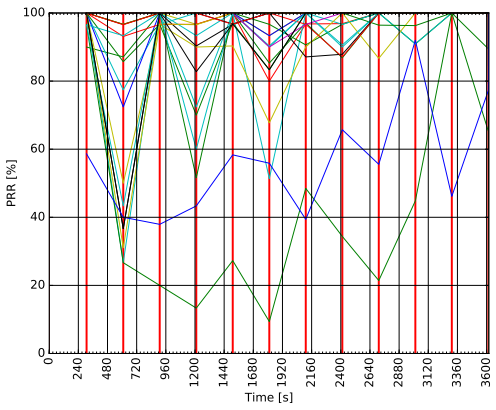


(b) The network over time with a `JL_WIFI4` jammer and OF0.

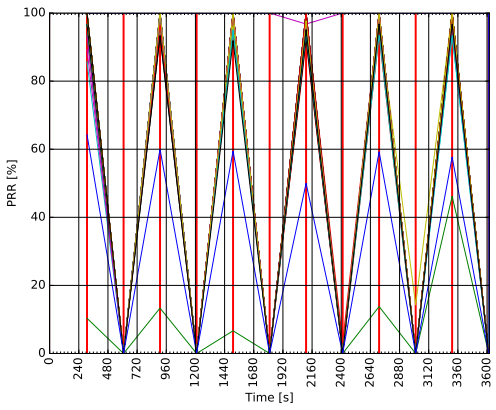
Figure 3.22: Collected events of the network from each node over the whole experiment time.

3.3.3 Performance of RPL with ETX and periodic jamming

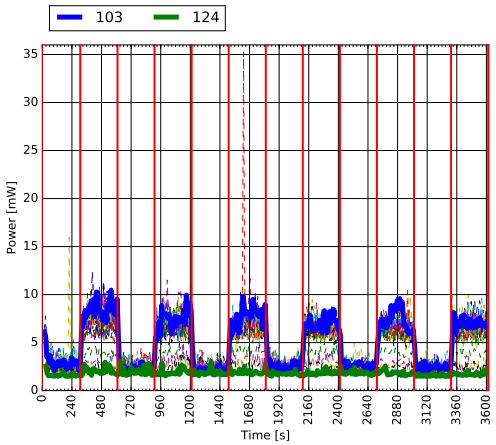
This section shows the performance of RPL in the presence of a periodic jammer. The latter interferes for 5 minutes followed by a 5 minute pause. The experiments are only run using `JL_WIFI4` and `JL_CONTN`, as lower interference levels have only little impact on the network (See results of the previous experiments in Table 3.2). The experiments are also only carried out using ETX, as previous experiments have shown its superiority over OF0. The goal of these experiments is to understand if and how the network "remembers"" a previous good configuration or rather "derives" an optimal parent selection (configuration) once interference manifests. Through periodically starting and stopping of the interference, the plots in Figure 3.23 as well as Figure 3.25 are derived.
The first visual impression indicates that the first network in Figure 3.25a improves over time. Figure 3.23a shows this in more detail. The average PRR of each node periodically decreases as the jammer is turned on, but the impact is reduced every time. After 60 minutes most nodes achieve a PRR of almost 100 %.



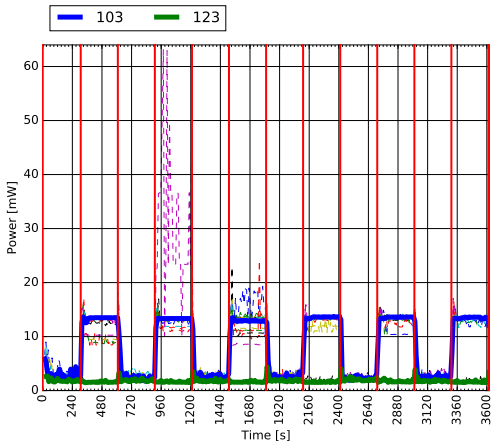
(a) Improving PRR with `JL_WIFI4`



(b) No improvements with `JL_CONTN`



(c) using `JL_WIFI4`

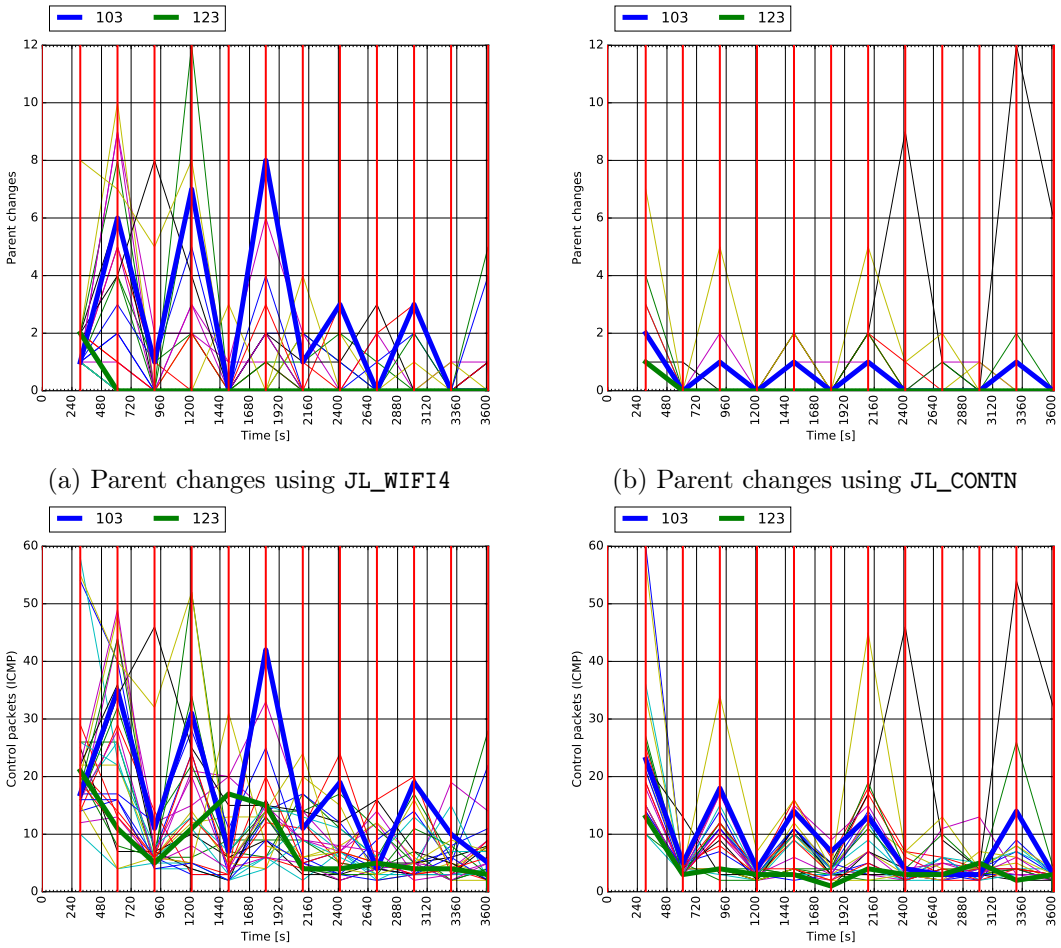


(d) using `JL_CONTN`

Figure 3.23: PRR and RX power consumption of the network with a periodic jammer.

In Figure 3.23b, the network is subjected to the constant carrier jammer (`JL_CONTN`). In this case, the PRR does not improve over time. While the jammer is active, the network is not able to communicate as shown in Figure 3.25b and thus it is also not possible to improve the parent selection. After the jammer is turned off, the network returns to its

previous state.
The alternating behaviour of the jammer is clearly visible in the power consumption. The energy only slightly reduces over time whereas the PRR increases.
In Figure 3.24 the number of parent changes as well as the number of ICMP control packets is shown. In the case of the continuous jammer (`JL_CONTN`) the numbers stay roughly the same, whereas with `JL_WIFI4` the number decreases even though the interference is present.



(a) Parent changes using `JL_WIFI4`

(b) Parent changes using `JL_CONTN`

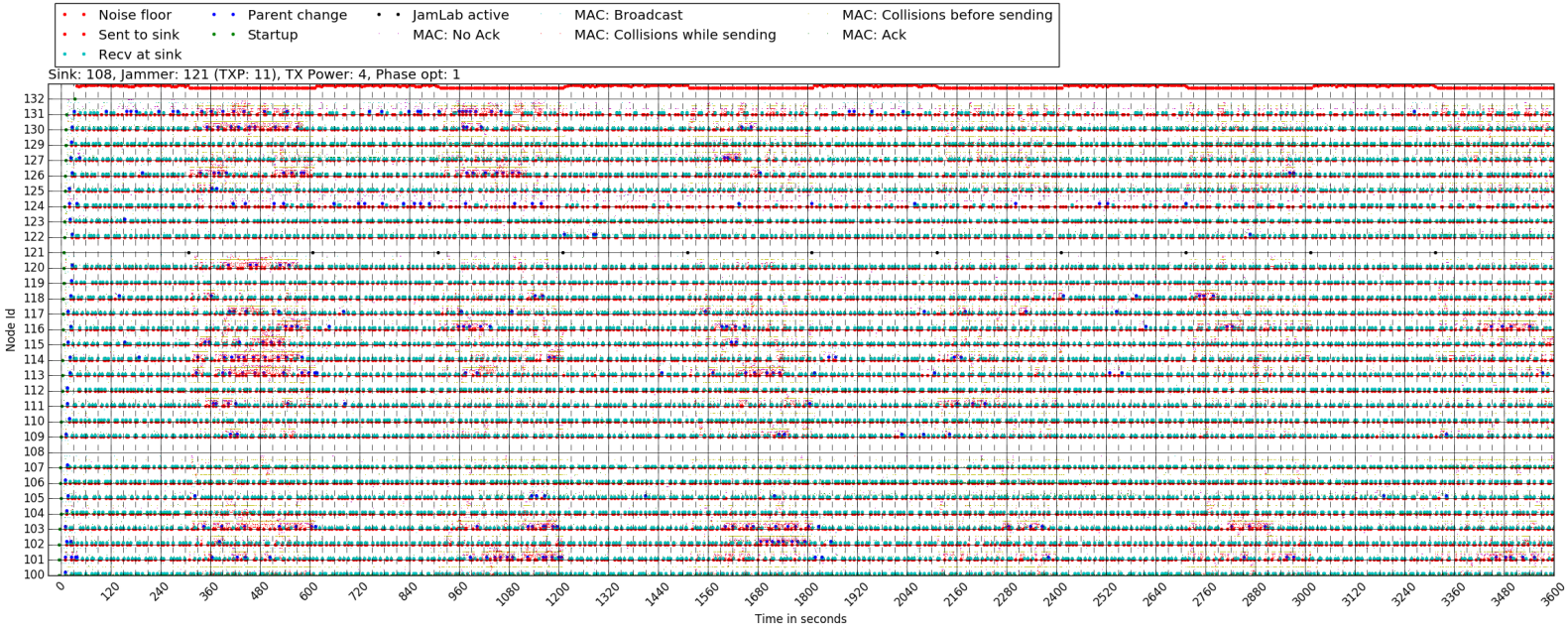
(c) Control packets using `JL_WIFI4`

(d) Control packets using `JL_CONTN`

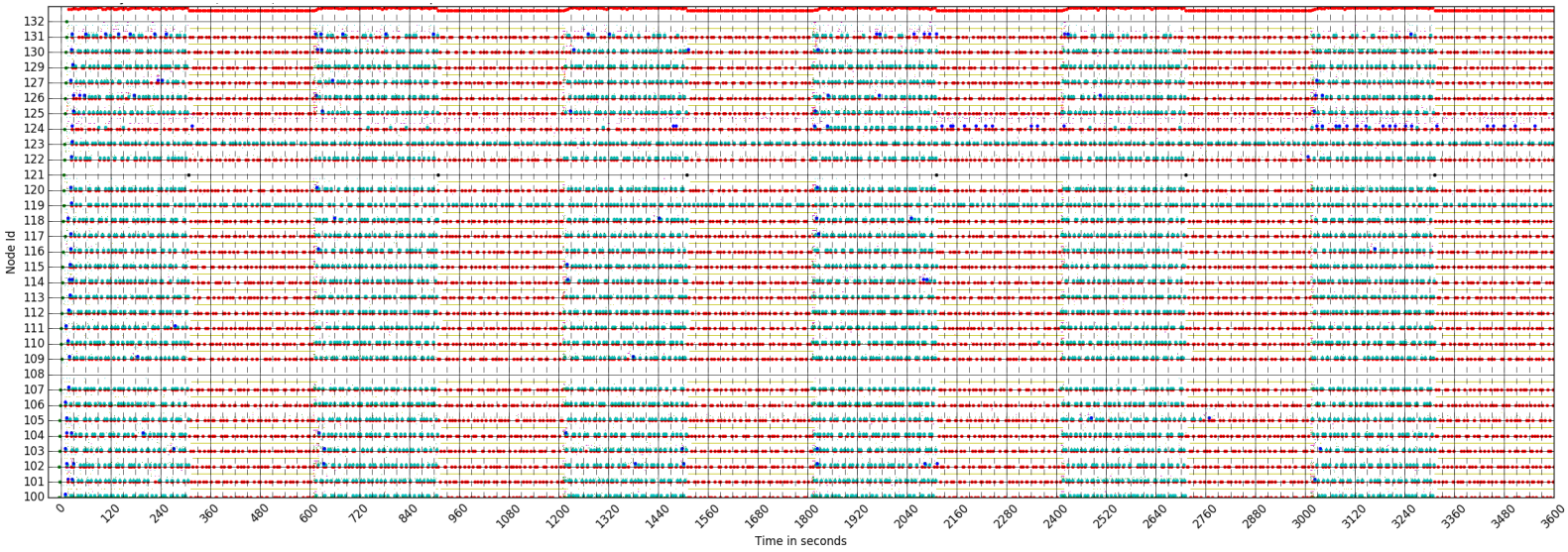
Figure 3.24: Number of parent changes and ICMP control packets in the network.

Lessons learned:

- *RPL tries to select a better topology over time, if at least some communication is possible (`JL_WIFI4`) but does not achieve a PRR of* 100 %.
- *If the communication is completely interrupted (`JL_CONTN`) the network returns to a similar state in terms of Energy consumption and parent switches as soon as the interference is gone.*
- *Power consumption on a "stable" configuration slightly decreases over time.*
- *Also in these experiments high interference (`JL_CONTN`) effectively blocks all communication as it can be seen in Figure 3.25b due to increasing the noise floor above CCA threshold.*



(a) The network over time with a periodic `JL_WIFI4` jammer.



(b) The network over time with a periodic `JL_CONTN` jammer.

Figure 3.25: Collected events of the network from each node over the whole experiment time.

3.3.4 Investigate changing CCA to escape interference

This section aims to verify that changing the CCA may help to escape interference. A network of two nodes is set up. The sender node **A** is selected so that the noise impact of the jammer **J** at this node is above the default CCA level of $-77\,\text{dBm}$. This blocks the nodes communication completely as soon as the jammer is turned on as shown in Table 3.3. The receiver node **B** is chosen so that the RSSI between them is above noise impact at both nodes.
To help find these nodes the Figure 3.26 has been created. It contains the noise impact at every node in the testbed as well as links between nodes if their RSSI is above $-65\,\text{dBm}$.

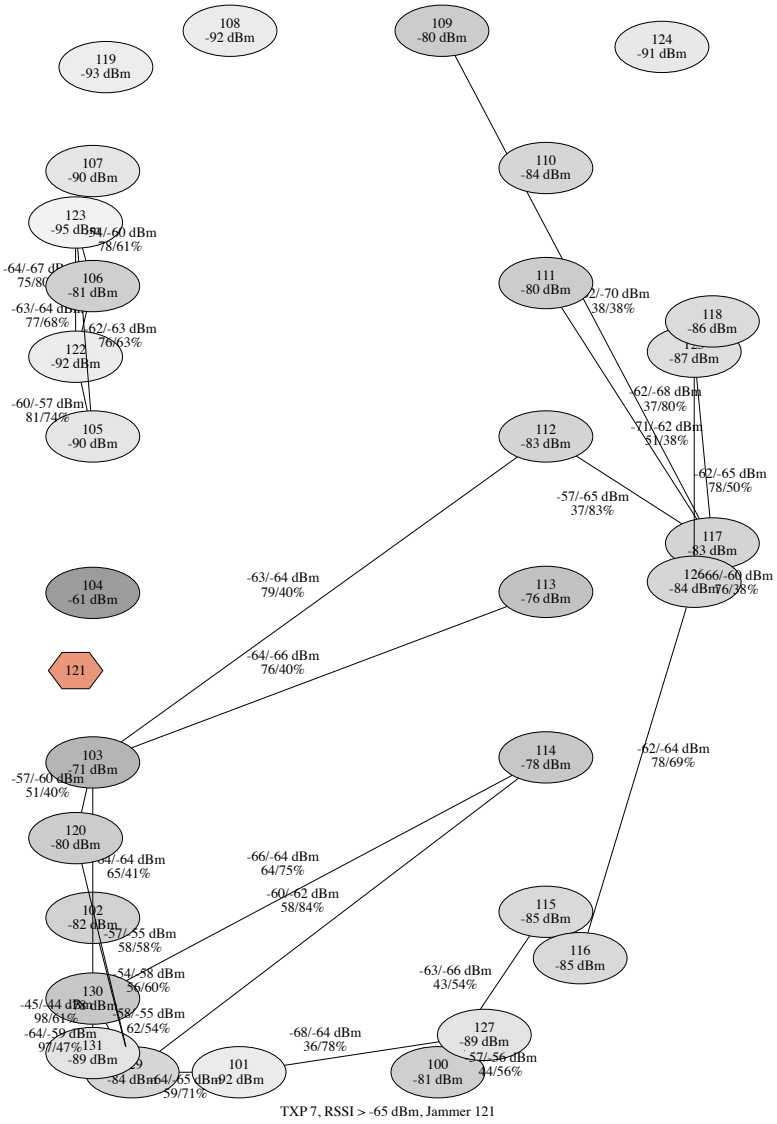


Figure 3.26: Testbed map with noise impact and links with an RSSI $>-65\,\text{dBm}$.

For these experiments the nodes 103 (**A**) and 112 (**B**) were chosen. The RSSI between them is at around −65 dBm. The noise impact at **A** is −71 dBm and at **B** it is −83 dBm. The results in Table 3.3 show that changing the CCA from −77 dBm to −68 dBm allows the nodes to communicate beside interference. Also the energy consumption is not affected by the noise any more and returns to almost the same value, as if no interference is present.

Jam Type	**CCA** [dBm]	**PRR** [%]	**RSSI** [dBm]	**Power RX** [mW]	**Power TX** [mW]	**Power CPU** [mW]	**Power LPM** [mW]
-	-77	100	-66	1.53	0.04	0.49	0.18
JL_CONTN	-77	0	-65	12.99	0.00	3.02	0.18
-	-68	100	-66	1.53	0.04	0.49	0.18
JL_CONTN	-68	100	-65	1.56	0.03	0.48	0.18

Table 3.3: Results of the CCA change validation experiments.

This shows that by changing only the CCA value an improvement to the reliability (less energy, higher PRR) in a noisy environment can be made.

Lessons learned:

- *Changing only the CCA to escape some types of radio interference is possible.*
- *Nodes with a CCA above the noise floor are able to send messages regardless of the noise.*
- *The energy consumption decreases if the CCA is changed to almost the same value as if no radio interfere is present.*

Chapter 4

Design and Implementation

The experimental campaign has shown a number of observations about the performance of RPL in a network subjected to radio interference. One of the observations is that the network completely breaks if a strong enough interference level (`JL_CONTN`) is present, as no message is sent.
Among others, the logging of `ContikiMAC` records many *collisions before sending*: as shown in Section 3.3.2, these collisions are generated by a failing CCA check occurring when the RSSI (noise floor) of the radio channel is above the default CCA threshold and the channel is thus considered permanently busy.

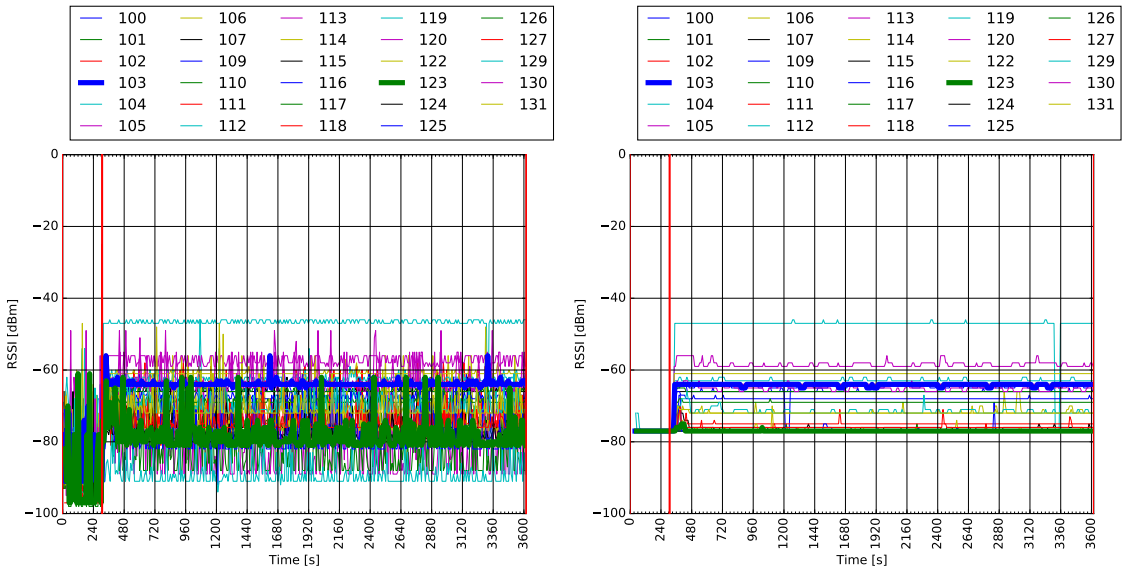
4.1 The role of CCA

To limit the number of collisions before sending, the aim is to increase the CCA threshold. This is however a "catch-22 dilemma". Increasing its value too high may lead to packet collisions, as this threshold is an integral part of the collision avoidance mechanism of the MAC layer. Setting the CCA threshold too low would block the transmitter continuously: this is currently the case if the jammer is active.
Increasing the threshold would also bring an advantage on the receiving side; as less false wake-ups result in lower energy consumption. If the threshold is set too high, however, no packets may be received, as the radio is turned off immediately after the periodic CCA. Setting the CCA threshold too low would introduce a higher number of false wake-ups and to a higher energy consumption.
These upper and lower limits need to be determined while the network is active by exploiting additional informations such as the RSSI values of the received packets as well as by periodically measuring the noise floor. The new CCA threshold will then be bound to these upper and lower limit constraints. This is a similar approach as shown by Boano et al. in [4] to mitigate the adverse effects of temperature in WSNs.

4.2 Design

The goal is to implement an automatic adaptive CCA threshold adjustment algorithm (AutoCCA). The approach proposed in this thesis periodically measures the noise floor and then sets the CCA threshold above this value. Measuring the noise floor should not

have an impact on the nodes operation. Although this cannot be entirely avoided, it is imperative to minimize energy consumption.
A high number of RSSI samples in a short period of time are required to get a good estimate. To reduce the data size, every measured RSSI value counts toward its array entry in an histogram-like data structure. The array index is the measured RSSI level and the value is the sum of its occurrences. This histogram implementation is already used in the `noise_analyzer` tool as discussed in Section 2.5.2.
The measured RSSI values are expected to have a baseline and some volatile outliers. The baseline is our value of interest as it is increased by the jammer. An example measurement is shown in Figure 4.1a. On the left hand side of the vertical red line the jammer is not yet turned on. The lower bound for the read values is at about −95 dBm as defined by the sensitivity of the CC2420 radio according to its datasheet [31]. As soon as the jammer is turned on (right hand side of the red vertical line) the lowest measured values are increased, providing a new baseline.
In Figure 4.1 nodes 103 and 123 are highlighted. Node 123 is an example for a node that is not affected by the jammer. Node 103 is affected and thus has an increased noise floor.



(a) without filtering

(b) with filtering

Figure 4.1: Example noise floor as measured before and after turning on the jammer. Node 103 is affected by the jammer and has an increased noise floor. Node 123 is only lightly affected and the noise floor stays below −77 dBm.

$$x_t = \text{find\_noise\_floor}(t) + \epsilon \tag{4.1}$$

$$x_t = \max\left(x_t, -77\,\text{dBm}\right) \tag{4.2}$$

$$\text{CCA}_t = \min\left(x_t, x_{t-1}, x_{t-2}, x_{t-3}, \ldots, x_{t-n-1}\right) + \beta \tag{4.3}$$

After measuring the noise floor, a safety margin ϵ is added as shown in Equation 4.1. As a default value, ϵ is set to 3 dB. This is the co-channel rejection value of the CC2420

radio [31]. If this is the best value for the present experimental setup will be determined experimentally in Section 5.2.
The noise floor is then limited to the default threshold of $-77\,\text{dBm}$ as shown in Equation 4.2. The default CCA threshold is already high enough to provide a good PRR in the testbed if no noise is present as shown in the Table 3.1. A CCA threshold lower than $-77\,\text{dBm}$ would increase the amount of false wake-ups and also allow a parent selection of a node with a lower RSSI and thus increasing the chance of failure under interference. On the other hand lowering would be needed if the network is not yet connected.
From the measured value x_t a filtered baseline CCA_t is calculated and used as a lower bound for the CCA threshold.
Filtering the measured noise floor is important to keep the system stable. As only the lower bound is of interest, the filter uses the minimum value of the last n samples as shown in Equation 4.3. For $n = 4$, the result is shown in Figure 4.1b.
An upper bound for the CCA threshold is not required, as packets with an RSSI below the noise floor have a high chance of corruption due to radio interference. Increasing the value too high will prevent the node from receiving any messages, but on the other hand also reduces its energy consumption due to reduced false wake-ups. Thus, defining an upper limit of the CCA threshold has no real benefit.
Adding another dynamic value β as shown in Equation 4.3 could be used to help selecting a better parent by forcing a reduced set of parents ($\beta > 0$) or by lowering the CCA threshold to ensure that the network is connected ($\beta < 0$). This feature is currently not implemented in this thesis. It would require to combine knowledge about the current network performance and the nodes parents. This would then also allow to keep the CCA threshold at a higher level even though when the noise reduces again.

4.3 Implementation

The implementation of the AutoCCA algorithm as described in Section 4.2 is shown in Listing 4.1. This function is called every 10 s in a Contiki OS process and measures the current noise floor (about 50 ms for 1000 measurements), determines a new filtered value, and updates the CCA threshold of the CC2420 radio. The function could also be implemented as a process on its own, but due to code size limitations and an already present process that executes every 10 s, a function has been chosen.

```
1  static void autocca(void) {
2      rtimer_clock_t now = RTIMER_NOW();
3      int8_t noise_floor_dbm = find_noise_floor();
4      now = (RTIMER_NOW() - now);
5
6      PRINTF("NoiseF m %d/%u, value: %d dBm\n",
7             now, RTIMER_SECOND, noise_floor_dbm);
8
9      #if (AUTO_UPDATE_CCA > 0)
10     {
11       static int8_t prev_noise_floor_dbm[3] = {-77, -77, -77};
12       int8_t current_cca;
13       noise_floor_dbm += AUTO_CCA_EPSILON;
14
15       if (noise_floor_dbm < -77)
```

```
16        {
17          noise_floor_dbm = -77;
18        }
19
20        prev_noise_floor_dbm[0] = prev_noise_floor_dbm[1];
21        prev_noise_floor_dbm[1] = prev_noise_floor_dbm[2];
22        prev_noise_floor_dbm[2] = noise_floor_dbm;
23
24        noise_floor_dbm = MIN(prev_noise_floor_dbm[0],
25                              MIN(prev_noise_floor_dbm[1],
26                                  MIN(prev_noise_floor_dbm[2],
27                                      noise_floor_dbm)));
28
29        NETSTACK_RADIO.get_value(RADIO_PARAM_CCA_THRESHOLD,
30                                 (radio_value_t *)&current_cca);
31
32        if (current_cca != noise_floor_dbm) {
33          PRINTF("NoiseF u %d to %d\n", current_cca, noise_floor_dbm);
34          NETSTACK_RADIO.set_value(RADIO_PARAM_CCA_THRESHOLD,
35                                   noise_floor_dbm);
36        }
37      }
38      #endif
39  }
```

Listing 4.1: Measure noise floor and update CCA threshold.

The implementation is straightforward: first `find_noise_floor()` measures the noise floor, then the filter is updated, and finally a new CCA threshold is set if it is different from the previous one. The code also measures the execution time to get a rough estimate of the duration of the noise floor measurement.

The implementation of `find_noise_floor()` is based on the sample code provided in the `noise_analyzer` (Section 2.5.2). It requires some changes to make it usable in applications where the RDC is turned on: it is important to correctly enable and disable the radio based on its previous state, as well as managing the locking mechanism of the CC2420 radio driver as messages might be received or sent asynchronously.

The current driver API does not provide functions to turn the radio on and to keep the lock until the radio is turned off again. The locking mechanism of the CC2420 radio driver uses static variables, which are not accessible outside of the `cc2420.c` driver file.

To extend the CC2420 driver, Contiki OS provides an easy mechanism: if a file with the same name as in the Contiki OS' sources exists in the project folder, it is preferred over the original one. This allows either to completely change the previous implementation or to extend it by including the existing `.c` file as shown in Listing 4.2. Including a `.c` file is quite unusual, but a good method in this case. Without adding this, the existing `cc2420.c` would have to be copied and changed (which would make it harder to keep the file updated).

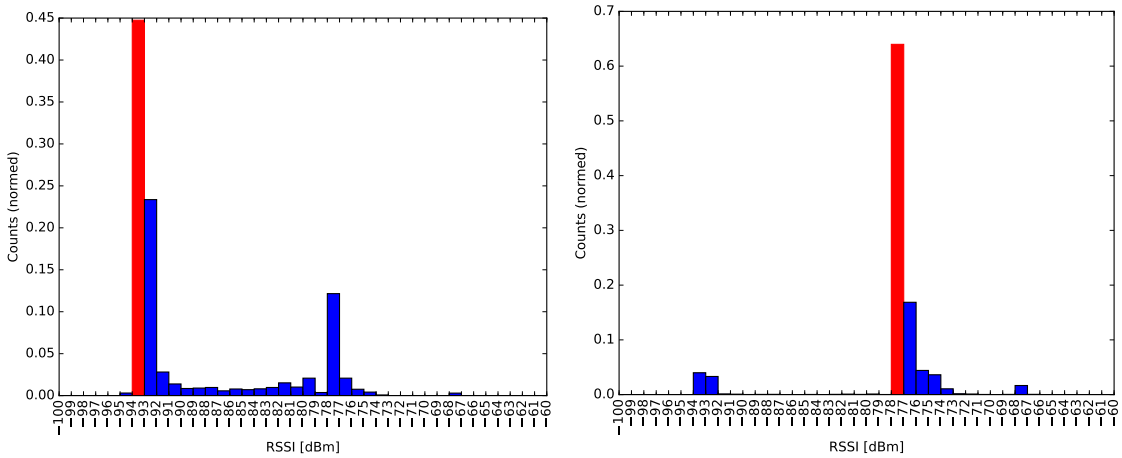
Two new functions are added to the CC2420 radio driver API. They split the already existing `cc2420_rssi()` function into two separate parts. The `_get_lock` function acquires the driver lock and turns the radio on if needed: it is called before starting the measurements. The `_release_lock` function releases the lock and turns the radio off if it has been off before calling the `_get_lock` function. It is called after finishing the measurements.

```
33 /* Include the original implementation and extend it.
34  * This allows access to statik variables.
35  * Contiki's build system  will prefer this file instead
36  * of the original one */
37 #include "../../contiki/dev/cc2420/cc2420.c"
38 #include "cc2420-extension.h"
39
40 static int radio_was_off_fast = 0;
41 int
42 cc2420_get_lock(void)
43 {
44   if(locked) {
45     return 0;
46   }
47
48   if(!receive_on) {
49     radio_was_off_fast = 1;
50     cc2420_on();
51   }
52
53   GET_LOCK();
54   wait_for_status(BV(CC2420_RSSI_VALID));
55   return 1;
56 }
57
58 void
59 cc2420_release_lock(void)
60 {
61   RELEASE_LOCK();
62
63   if(radio_was_off_fast) {
64     cc2420_off();
65   }
66 }
```

Listing 4.2: Extended `cc2420.c` in the project folder.

The `find_noise_floor()` function supports multiple compile time configuration options, such as the number of samples or how the histogram is interpreted. One method to interpret the histogram is using the statistical mode. The mode returns the value with the highest number of occurrences. Using the highest measured RSSI is also possible and will be used for the evaluation.
Given the RSSI distribution of `JL_WIFI4`, using the highest measured RSSI yields good results. Another option would also be to use the 95 percentile: this would reduce some high outliers with low occurrences, but filtering the measurements is still required.

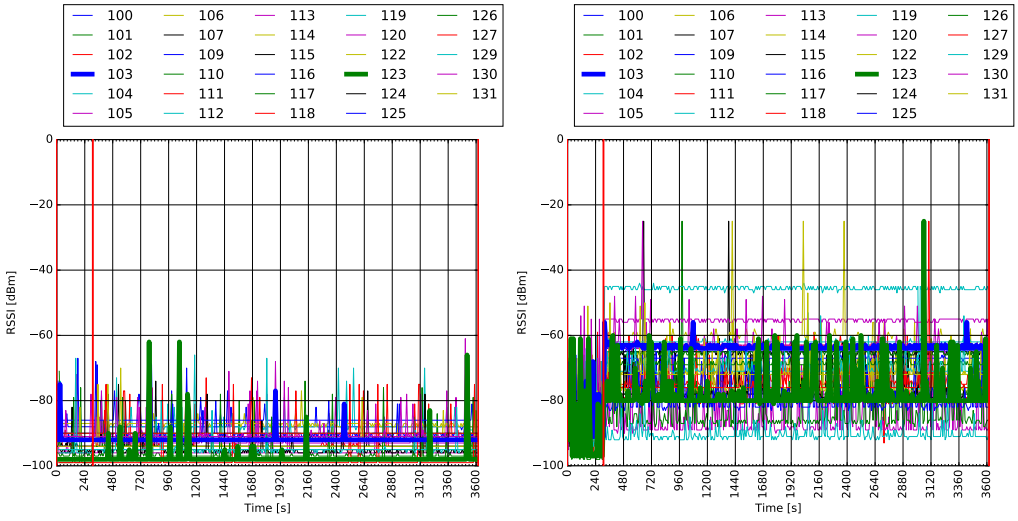
Figure 4.2 shows the histogram of a single node summed over the whole experiment time while interference of the types `JL_WIFI4` and `JL_CONTN` are present. The mode is highlighted in red. In the case of `JL_WIFI4` (Figure 4.2a), the mode returns the sensitivity of the radio (−94 dBm). This value is not helpful as the maximum noise generated by the interferer at this node is around −78 dBm. In the case of `JL_CONTN` (Figure 4.2b), using the mode would work, as the interference is present almost all the time.



(a) using `JL_WIFI4` (b) using `JL_CONTN`

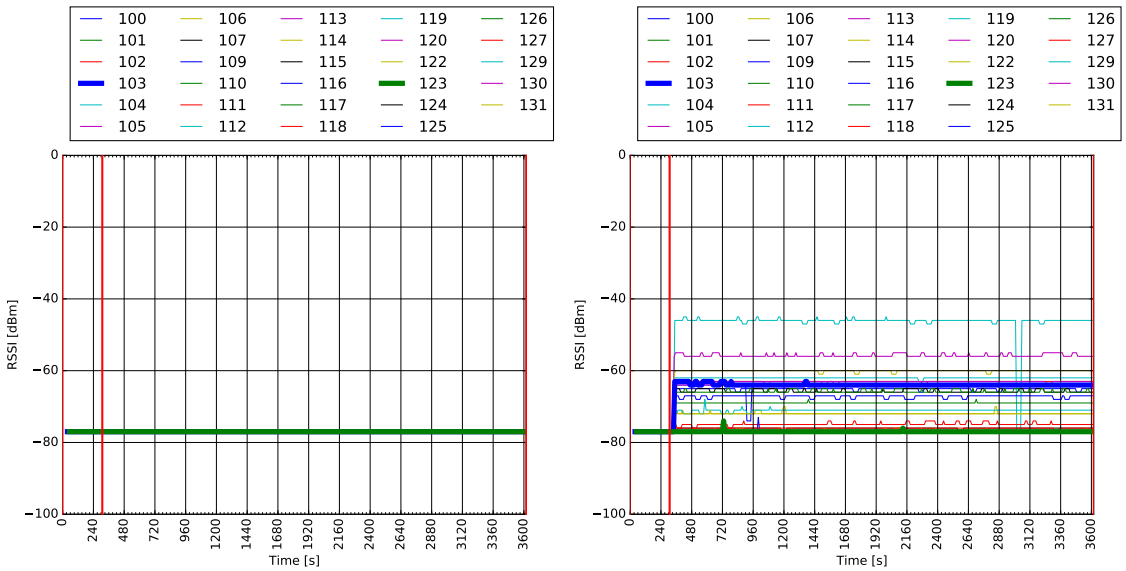
Figure 4.2: Histogram of the RSSI measurements on a single node.

Looking at the RSSI value selected from the histogram in a diagram over time as shown in Figure 4.3a the use of the statistical mode is not suitable. Even though the jammer is turned on, no increase in the RSSI is observable. Using the maximum value as shown in Figure 4.3b shows a clear "jump" as soon as the jammer is turned on. The high spikes are recorded if the measurement happens while another node is sending a message.
Applying the filtering method previously described to the maximum RSSI value provides a stable result as shown in Figure 4.4b. Using the mode, the filtered value always stays at the lower limit of −77 dBm (Figure 4.4a).



(a) using the mode (b) using the maximum

Figure 4.3: Statistical method used while measuring the noise floor of `JL_WIFI4`.



(a) using the mode

(b) using the maximum

Figure 4.4: Filtered noise floor measured while `JL_WIFI4` is present.

4.4 Limitations

The design and implementation presented in this sections has some limitations regarding the lower limit of the new calculated CCA threshold.
The new CCA threshold solely depends on the measured interference. Setting the CCA threshold too low will result in a network where the nodes prefer longer distance links which are of bad quality in terms of signal strength (but reduce the number of hops).
This requires that the algorithm uses a lower limit (see Equation 4.2) that has to be known a priori. This limit must be selected such that the network is connected if no interference is present. In the present experimental setup, using Contiki OS' default CCA threshold of $-77\,\mathrm{dBm}$, everything works as expected. If the network is not yet connected due the physical node distance and set transmission power, this value must be lowered manually until the network is connected.
Further improvements to the energy consumption of the node could be achieved if the CCA threshold would be increased as long as the network is connected. With the current solution this is not possible, as no knowledge of the network performance is used, and as the CCA threshold is always lowered again to match the noise floor but kept above a limit plus ϵ.

The present solution only provides a method to escape interference and to reduce the energy consumption if interference is present. Other improvements achievable by automatically adjusting the CCA threshold (limiting or increasing the set of parents to reduce energy or to increase the PRR) are not yet explored or implemented.

Chapter 5

Evaluation

This chapter evaluates the improvements to `ContikiMAC` described in Chapter 4. The goal of this chapter is to show how the performance of the network improves if the CCA threshold is dynamically changed using the proposed AutoCCA algorithm. The experimental setup is the same as the one used in the previous experiments. The results of the experiments presented in the experimental campaign in Chapter 3 are compared to similar experiments in which the CCA threshold is changed automatically.
The following experiment configurations are investigated:

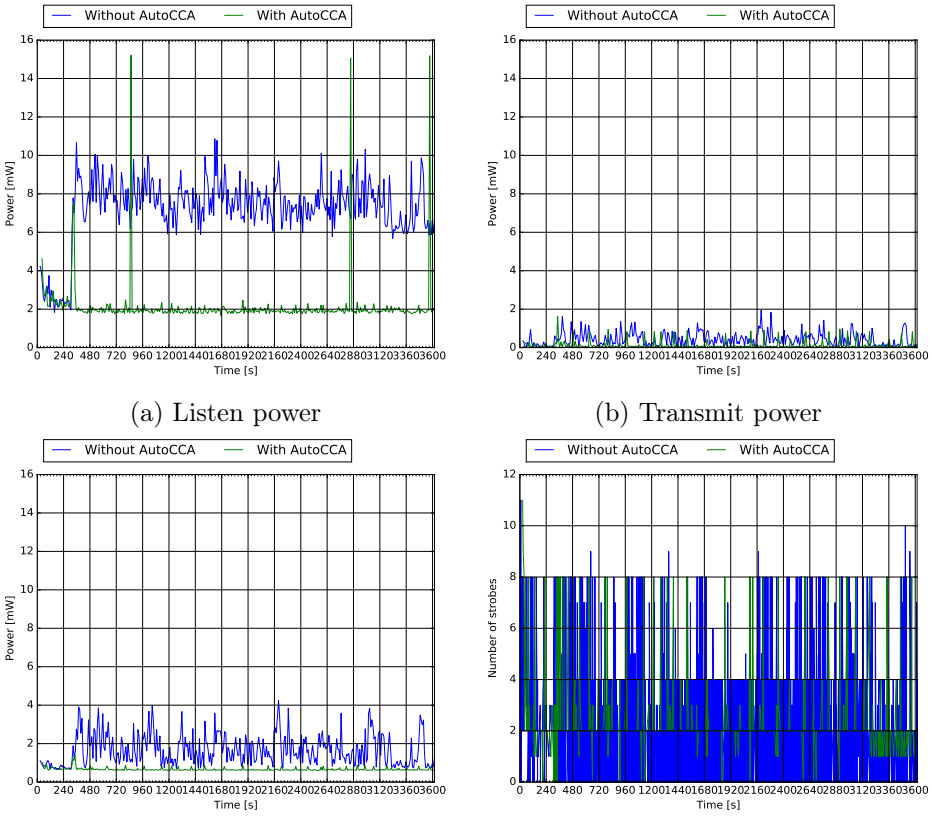
- No added radio interference;
- `JL_WIFI4` interference continuously on;
- `JL_CONTN` interference continuously on.

Each experiment uses the same nodes, a duration of one hour, and ETX as objective function. For these experiments, OF0 and light interference are not run, as ETX always performed better than OF0 (see Section 3.3.2), and as `JL_WIFI1` and `JL_WIFI2` only had a small influence on network performance (see Table 3.2).

Jam Type	**AutoCCA**	**Nodes**	**PRR** [%]	**RSSI** [dBm]	**Power RX** [mW]	**Power TX** [mW]	**Power CPU** [mW]	**Power LPM** [mW]	**Power Total** [mW]	**Parent changes/node**	**Nodes PRR >90%**
`JL_NOINT`	NO	30	94	-73	2.30	0.16	0.81	0.18	3.45	4.000	27
`JL_NOINT`	YES	30	97	-74	2.25	0.10	0.76	0.18	3.29	3.828	28
`JL_WIFI4`	NO	30	86	-72	6.24	0.33	1.41	0.18	8.16	13.241	18
`JL_WIFI4`	YES	30	89	-72	2.33	0.24	0.79	0.18	3.53	4.241	25
`JL_CONTN`	NO	30	9	-69	10.98	0.04	3.32	0.18	14.52	0.517	2
`JL_CONTN`	YES	30	61	-67	2.83	0.58	0.88	0.18	4.47	10.966	15

Table 5.1: Evaluation results with and without enabled AutoCCA.

Table 5.1 shows the results of the repeated experiments, while also the automatic CCA threshold change as described in Chapter 4 is enabled. The results of the previous experiments are included as well.
The results of the experiment without interference shows that the added noise floor measurements do not decrease the nodes performance (almost same PRR and total power consumption).
Packet reception rate. It can be noted that in the case of `JL_WIFI4`, only a small improvement of the PRR is visible (from 86 % to 89 %). But the number of nodes with a PRR above 90 % now almost covers every node used (from 18/29 to 25/29). In the case of `JL_CONTN` the improvements are more evident. Starting from a completely broken network with only two nodes achieving a PRR above 90 %, the network manages to form a network reaching about 15/29 nodes. This results in an increase in the average PRR from 9 % to about 61 %.
Power consumption. The energy consumption for `JL_WIFI4` is significantly lowered. The value decreases from 8.16 mW to 3.53 mW, an improvement of 56 %. For `JL_CONTN` the average total power consumption is reduced from 14.52 mW to 4.47 mW, an improvement of 69 %. The power consumption of `JL_WIFI4` and `JL_CONTN` hence almost reaches the value recorded when no interference is present (3.29 mW).
Even though not all nodes manage to connect to the network, the availability is significantly increased, as the batteries will last approximately three times longer.
Figure 5.1 shows the power usage of a single node with and without enabled dynamic CCA threshold update. It can be observed that the listen power consumption stays at a low level of about 2 mW if AutoCCA is enabled. Without the improvements the energy increases to about 8 mW, as soon as the jammer is turned on.



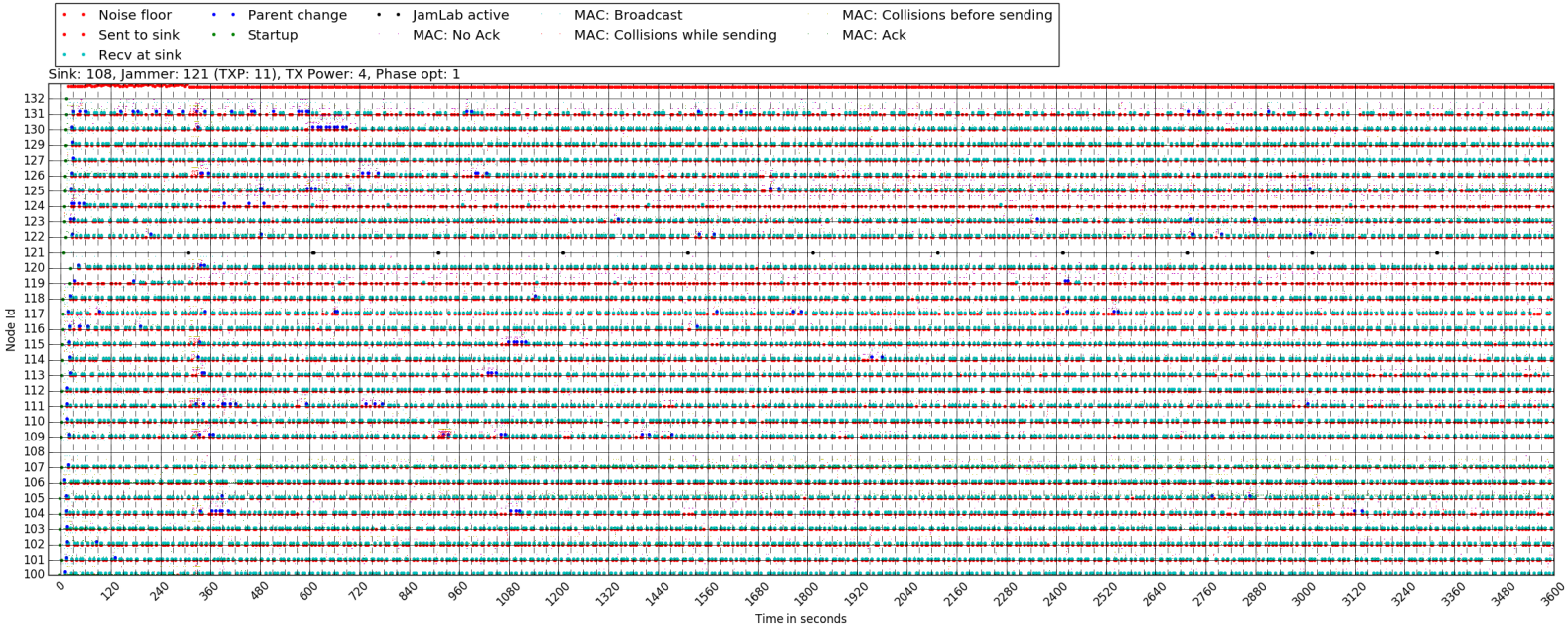
(a) Listen power

(b) Transmit power

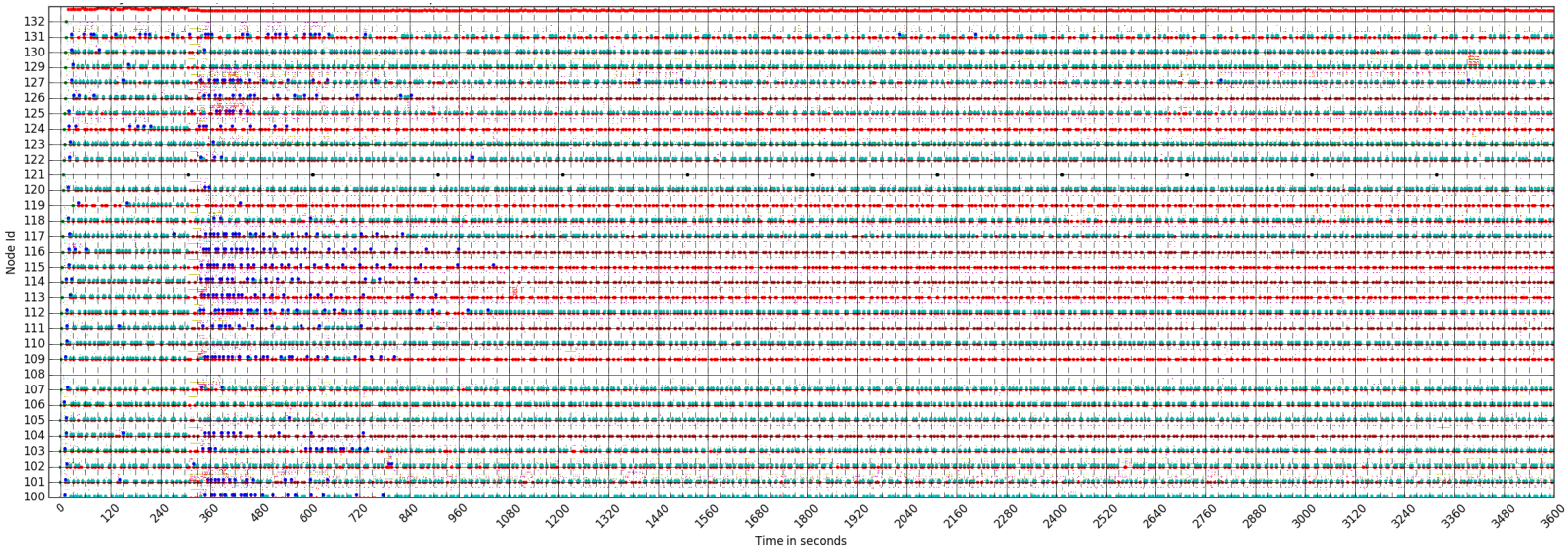
(c) CPU power

(d) Number of strobe packets

Figure 5.1: Power usage and number of strobe packets with and without AutoCCA.



(a) The network over time with a continuous `JL_WIFI4` jammer.



(b) The network over time with a continuous `JL_CONTN` jammer.

Figure 5.2: Collected events of the network from each node over the whole experiment time with AutoCCA enabled.

Parent changes. In the case of `JL_WIFI4` a lot less parent changes per node are counted (from 13.24 down to 4.24), almost reaching the value observed when no interference is present (3.8). The network is way more stable. In the case of `JL_CONTN`, the number of parent changes per node is increased from 0.51 to 10.96. In the previous experiment the nodes communication was blocked and thus no parent change could be performed. The network is still not as stable as when `JL_WIFI4` is active, given that not every node is able to communicate.

Figure 5.2 shows the time graph of the whole network with AutoCCA enabled. There are only some collisions before sending visible after the jammer is turned on. As soon as the CCA threshold is changed no more collisions are recorded (compare with Figure 3.21).

Lessons learned:

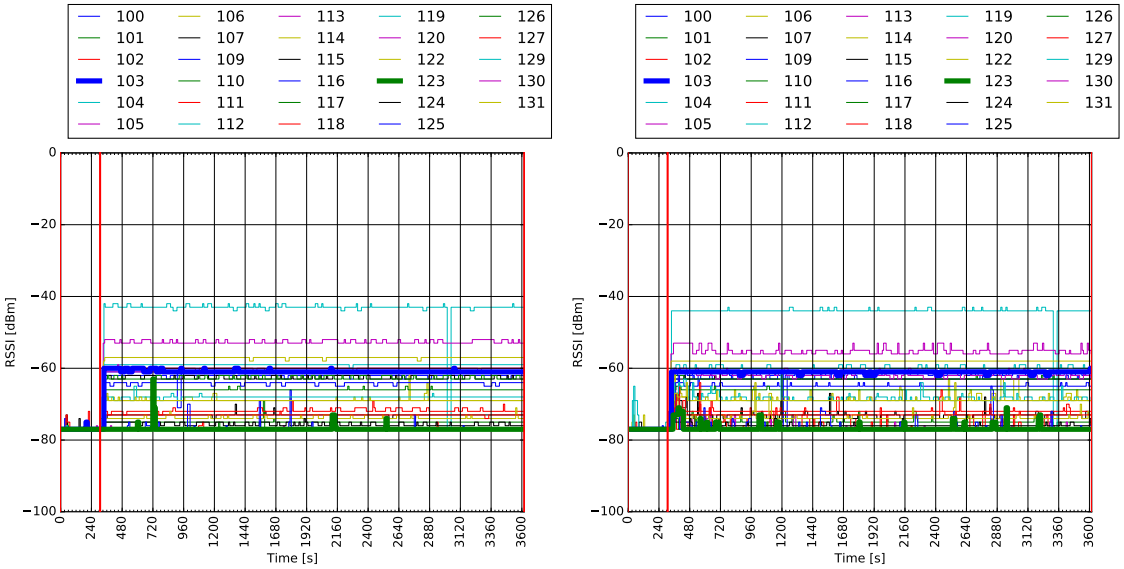
- *Automatically changing the CCA threshold lowers the energy consumption in the presence of interference notably.*
- *The added noise floor measurements have only little impact on the node. Measuring every* 10 s *for* 50 ms *is sufficient.*
- *The PRR increase from* 9 % *to* 61 % *for the continuous carrier jammer (`JL_CONTN`), shows indeed that changing the CCA threshold in high interference situation is useful.*

5.1 Monitoring automatic CCA threshold updates

To observe and monitor the CCA threshold updates, every change is logged and can be plotted. The results in a network with interference are shown in Figure 5.3.
As soon as the jammer is turned on, each node adjusts (increases) its CCA threshold according to the design and implementation from Chapter 4. Each node selects a different CCA threshold, as the impact of the jammer depends on their physical position (see Section 3.2.3).
Figure 5.4 shows the CCA threshold updates if a periodic jammer is present. The interference level is toggled after five minutes. As soon as the jammer is turned on, the CCA threshold is increased to adapt to the new noise floor. After the jammer is turned off the CCA threshold returns to its previous value. Keeping the CCA threshold at a higher level would allow for an even more reduced energy consumption, but requires to combine knowledge like the number of available parents from RPL. This has not yet been implemented, but would also be beneficial if no interference is present.

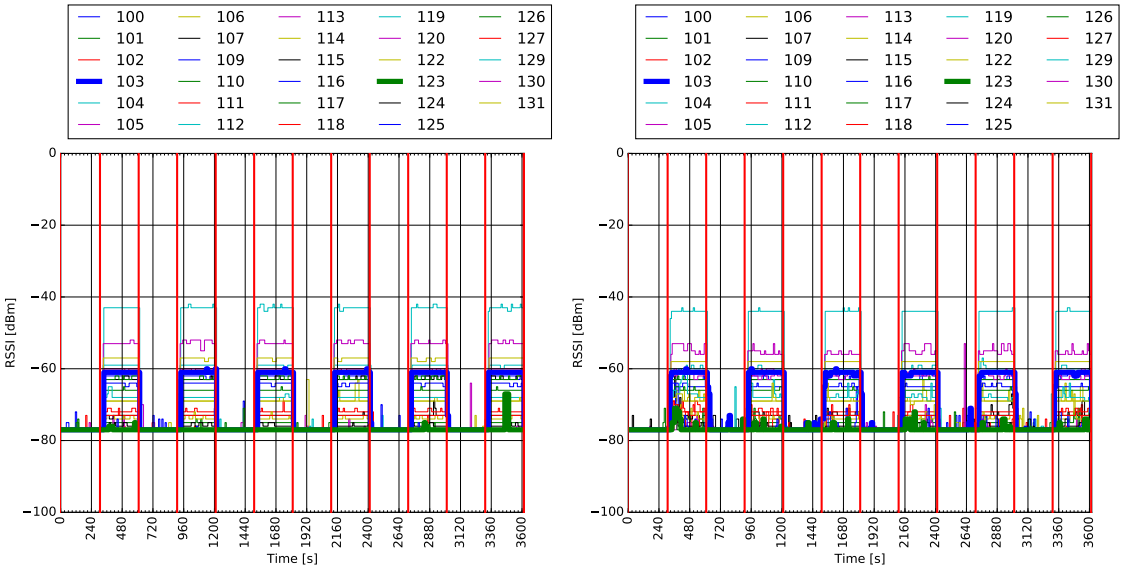
Lessons learned:

- *Rapid changes of the noise floor update the CCA threshold quite fast (max* 40 s *when using the last four measurements and an update interval of* 10 s*).*
- *Using the minimum value of the last four measurement as the sole filter is sufficient for a quite stable CCA threshold.*
- *The algorithm adapts correctly and repeatable to noise floor changes.*



(a) using `JL_WIFI4` (b) using `JL_CONTN`

Figure 5.3: Updates of the CCA threshold (continuous jammer).



(a) using `JL_WIFI4` (b) using `JL_CONTN`

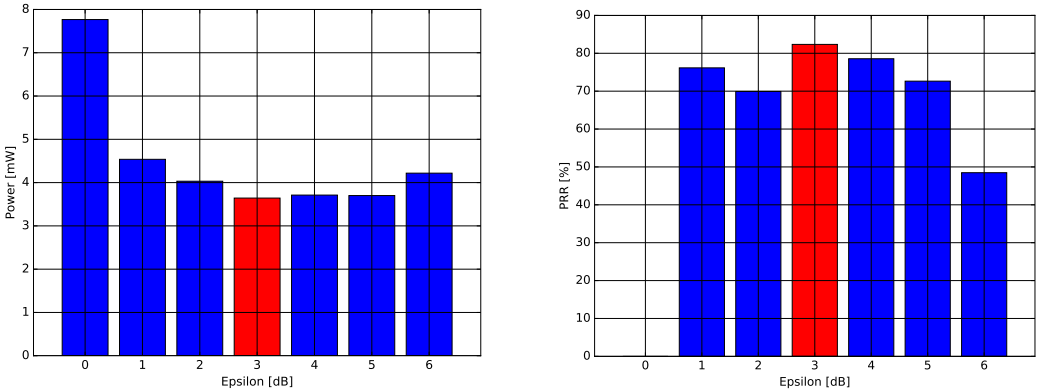
Figure 5.4: Updates of the CCA threshold (periodic jammer).

5.2 Validation of ϵ

The CCA threshold is set to the measured noise floor plus a fixed ϵ as shown in Equation 4.1 and discussed in Section 4.2. For the experiments summarised in Table 5.1 an $\epsilon = 3\,\text{dB}$ has been selected. The seven experiments in this section validate that choosing 3 dB as ϵ is a good choice. Table 5.2 compares seven values of ϵ from 0 to 6 dB and shows that the experiment using 3 dB yields the best result. The experiment with $\epsilon = 3\,\text{dB}$ has indeed the highest PRR and the lowest total energy consumption (3.64 mW).
Please note that the highest PRR in this table is not exactly the same as in Table 5.1 (82 % vs. 86 %). Every experiment run yields slightly different results as a new different RPL tree is created. Furthermore, also the environment surrounding the nodes changes over time. Other students are using the room in which the testbed is deployed. This may results in moved monitors, chairs as well as in temperature changes that affect radio propagation and reception [5].
The experiments in this validation are run back to back at night to ensure that the effects of a changed environment are minimized.
Figure 5.5 shows the total power consumption and the PRR summarised in Table 5.2. The figure shows that an ϵ of 3 dB yields the best result. It has the lowest total power consumption of 3.64 mW as well as the highest PRR of 82 %. Using an ϵ of 0 does not work at all, giving a PRR of 0 % and it has also the highest power consumption with a value of 7.7 mW.
If the ϵ is too high (6) the network is not connected any more. Packets are dropped as `ContikiMAC`'s radio duty cycling does not keep the radio on to receive a packet.

ϵ [dB]	**Jam Type**	**AutoCCA**	**Nodes**	**PRR** [%]	**RSSI** [dBm]	**Power RX** [mW]	**Power TX** [mW]	**Power CPU** [mW]	**Power LPM** [mW]	**Power Total** [mW]	**Parent changes/node**	**Nodes PRR >90%**
0	`JL_WIFI4`	YES	30	0	-66	5.05	1.22	1.31	0.18	7.77	12.310	0
1	`JL_WIFI4`	YES	30	76	-71	3.02	0.41	0.93	0.18	4.54	11.310	17
2	`JL_WIFI4`	YES	30	70	-70	2.56	0.43	0.86	0.18	4.03	10.690	15
3	`JL_WIFI4`	YES	30	82	-70	2.33	0.31	0.83	0.18	3.64	8.724	23
4	`JL_WIFI4`	YES	30	79	-71	2.35	0.34	0.84	0.18	3.71	6.862	19
5	`JL_WIFI4`	YES	30	73	-72	2.30	0.38	0.83	0.18	3.70	9.552	17
6	`JL_WIFI4`	YES	30	48	-71	2.52	0.60	0.91	0.18	4.22	15.448	8

Table 5.2: Results as a function of ϵ.



(a) Power [mW], lower is better

(b) PRR [%], higher is better

Figure 5.5: Power consumption and PRR of different values of ϵ.

Lessons learned:

- *Choosing the right ϵ is important and has high influence on the network performance.*
- *A "bad" ϵ highlights the dilemma described in Section 4.1. Either no messages are send as the radio is considered blocked all the time ($\epsilon = 0\,\text{dBm}$) or only few messages are received as the radio is not kept awake ($\epsilon = 6\,\text{dBm}$).*

5.3 Manual verification of results

The experimental results arise further questions: is it possible to improve the network performance further by adjusting the CCA? The connectivity graph from Figure 3.7 provides a theoretical optimal solution. If the connectivity graph shows that a node would be able to communicate with another node where the link has a RSSI above the noise floor, the node should selected that node as a parent.
Table 5.3 shows the result of such an investigation. Most nodes are not improvable, as no other node that can provide a better link RSSI exists. Node 131 would be improvable, but, due to its physical location, even without noise it does not manage to reliably connect to the RPL tree.
A status is considered "improvable", if another node exits which has an RSSI above the noise floor including the $\epsilon = 3\,\text{dB}$ margin as used by the AutoCCA algorithm. Nodes might be listed with an "ok" even if their noise floor is above the best neighbour's RSSI, as long as their PRR is high.
None of the nodes except 131 is improvable in the current setup. This leads to the assumption that the currently implemented AutoCCA algorithm finds a good solution to improve the PRR, which can not be improved further without increasing the algorithms complexity significantly. For example, nodes 119 and 124 could probably be improved by implementing the β variable from Equation 4.3 to also lower the CCA threshold below

−77 dBm.

Node	Best neighbour RSSI [dBm]	Noise Floor JL_WIFI4 [dBm]	PRR with JL_WIFI4 [%]	Status	Noise Floor JL_CONTN [dBm]	PRR with JL_CONTN [%]	Status
100	-70	-65	100	ok	-66	91.82	ok
101	-68	-77	99.7	ok	-77	82.48	ok
102	-61	-77	99.7	ok	-77	96.34	ok
103	-60	-64	99.09	ok	-64	92.42	ok
104	-66	-46	92.99	ok	-47	0	not improvable
105	-62	-77	99.09	ok	-77	98.78	ok
106	-59	-72	100	ok	-72	99.7	ok
107	-62	-76	100	ok	-77	97.88	ok
109	-66	-64	93.33	ok	-65	3.64	not improvable
110	-69	-77	99.39	ok	-77	99.09	ok
111	-62	-63	91.54	ok	-64	5.76	not improvable
112	-62	-62	100	ok	-63	92.97	ok
113	-70	-56	96.64	ok	-59	0	not improvable
114	-64	-60	97.26	ok	-61	1.82	not improvable
115	-72	-66	95.74	ok	-66	0	not improvable
116	-70	-67	98.48	ok	-68	0.3	not improvable
117	-69	-66	93.96	ok	-66	81.76	ok
118	-69	-75	98.79	ok	-75	93.31	ok
119	-78	-77	6.06	not improvable	-77	0	not improvable
120	-59	-63	98.79	ok	-64	98.17	ok
122	-60	-77	97.58	ok	-77	96.36	ok
123	-62	-77	96.97	ok	-77	99.7	ok
124	-79	-77	4.24	not improvable	-77	0	not improvable
125	-62	-77	93.01	ok	-77	92.73	ok
126	-67	-69	86.36	ok	-69	2.43	not improvable
127	-70	-76	99.7	ok	-76	82.62	ok
129	-54	-71	99.7	ok	-72	99.4	ok
130	-50	-64	95.45	ok	-66	99.09	ok
131	-50	-72	60.91	improvable	-72	52.12	improvable

Table 5.3: Nodes with a check if their parent is theoretical improvable.

Lessons learned:

- *The manual verification shows, that the current algorithm finds a good solution.*
- *Only one node would be improvable in the current setup using the presented AutoCCA algorithm.*
- *Increasing the complexity by implementing the β variable from Equation 4.3 might still improve the performance.*

Chapter 6

Related work

Reliability of WSN and RPL is a widely covered topic in research. This section provides an excerpt of some of the most important works covering RPLs performance that can be found in the literature. First RPL is compared to CTP in [23], where RPL is found superior. Further the performance of RPL is investigated while subjected to wireless interference [15] where problems are detected, but they are mostly attributed to the lower link layer protocols. It follows a study to evaluate three OFs with the result that no single OF exists yet which fits every application [19].
The next paper [29] presents AEDP an adaptive CCA protocol. Boano et al. show in the paper [4] that a dynamic CCA threshold is beneficial in the case of on-board temperature variations. Both papers supports the need and effectiveness for an adaptive CCA algorithm.

Nguyen Thanh Long et al. provide a *comparative performance study of RPL in WSNs* in their paper in [23]. They compare Collection Tree Protocol (CTP) [13] and RPL using `ContikiMAC` and `X-MAC` only in a simulated environment in Cooja, Contiki OS' network simulator. The authors show that the performance of CTP in terms of PRR and energy consumption is better in small networks (9 nodes) compared to when using RPL. For larger networks (25 to 49 nodes) and growing data traffic RPL outperforms CTP.
The experiments use up to 49 nodes laid out in a grid with the sink in the center or on the edge. Each node sends 1 to 4 packets per minute over a simulation time of 20 minutes. The tests are repeated 30 times with different random seeds. As metrics for evaluation the authors used the PRR, energy consumption per packet, number of loop packets, number of switching parents and the convergence time. In contrast to these metrics, this thesis focuses mainly on PRR and the total energy consumption.
Nguyen Thanh Long et al. first experiments show that using `ContikiMAC` the network converges 18 times faster than using `X-MAC` (605 s vs. 33 s) tested in a network of 25 nodes. Also the radio duty cycle of `ContikiMAC` is in their setup 10 times lower than `X-MAC`. Therefore this paper shows that the link layer protocol used has a huge influence on the performance of RPL.
The next interesting result show that using a node in the center as a sink gives the best performance. The authors achieve a PRR of about 96 % using a center sink and 25 nodes, while the PRR for a sink on the edge is only 87 %. In this thesis, the sink is always on the edge as shown in Figure 3.2 while still achieving a PRR of about 94 % as shown in Table 3.1. The nodes have a fixed distance of 10 m, while in this thesis experimental setup

the nodes are positioned with a distance between 0.5 m to 2 m. But the TXP is limited to still enforce multiple hops.
In their experiment Nguyen Thanh Long et al. increased the number of packets sent from 1 packet per minute to 4 packets per minute. Using RPL the PRR stayed high at close to 95 %. Using the higher data traffic, the PRR for CTP dropped to about 42 % while also increasing the number of loop packets from about 5 to more than 50 per minute. For RPL the number of loop packets stayed constantly below 5 packets per minute. To stress the network a bit more, this thesis uses about 6 packets per minute.
The tests of Nguyen Thanh Long et al. did not include any (external, radio-) interference introduced and the test were only performed in simulation but these results show again, that RPL is a good choice to be used as a routing protocol for data collection.

The article [15] of Dang Han et al., instead, investigates the *performance of RPL under wireless interference* explicitly. They build a network of 23 sensor nodes in a computer lab to monitor the energy used by each computer. Four types of interference are created and the networks performance is monitored. The lowest interference type used is IEEE 802.11 on a non overlapping radio channel. The second interference type is "normal" IEEE 802.11 traffic on an overlapping radio channel. As a third interference type, the authors deploy two nodes programmed with a high data transmission rate on the same 802.15.4 channel, as well as a IEEE 802.11 access point (mixed interference). The fourth interference type is created by generating a continuous 20 Mbps traffic between a IEEE 802.11 AP and a notebook.
The authors setup consists of PowerNet nodes running the TinyOS' RPL implementation using OF0 as OF. The PowerNet nodes also employ the IEEE 802.15.4 compatible CC2420 radio. This is the same radio as used in this thesis.
The nodes send a packet every 2 seconds plus some extra meta data information about the network every minute. Each of the experiments were run for 10 hours. As metrics, the nodes are recording the PRR, path ETX, control overhead ratio, and parent change rate, as well as a new metric called the excess parent changes (percentage of parent changes that did not improve the nodes rank).
The authors also highlight that packet loss in critical applications such as home monitoring is not tolerable. The results show that the average PRR drops from around 100 % with low interference to 92 % under normal interference. By increasing the interference level to the third type, the PRR drops to an average of 75 %. Using the highest interference level the PRR drops even below 10 %, making the network unusable. The measured control overhead ratio is at about 0.2 for low interference and increases to almost 1.0 for the high interference.
Further investigation has shown that the bad performance under heavy interference as found in typical smart home environments is not solely RPL's fault. The authors state that a better coordination between the networking layers is required to improve the performance. Some suggestions in the article are "more agile physical layer, less aggressive MAC, transport discrimination between different types of losses, moderation of data rates at the application" [15]. This provides some justification to the investigations and improvements to `ContikiMAC` made in this thesis.

Oana Iova et al. have investigated the *stability of RPL under wireless interference* [19].

They implemented and evaluated three different RPL objective functions (MinHop / OF0, ETX and LQI) in a simulation. The simulation setup is built using Contiki OS' RPL implementation and run in the WSNet simulator using 100 nodes. The results are averaged over 20 simulations using different random topologies. The nodes did not use any radio duty-cycling to reduce the influence of the lower link-layer protocols. This is a good idea if only the differences between the OFs are of interest, as this thesis shows that the link layer protocol has a huge influence on the network performance.
Oana Iova et al.'s results show, that each of the presented OFs has its own drawbacks. Using MinHop / OF0 gives better results in terms of latency and in energy consumption than LQI and ETX. OF0 minimizes the number of hops, thus less packets need to be forwarded, reducing both the energy consumption and the latency for the cost of more dropped packets (lower PRR) for nodes further away from the sink. Using LQI has the best results in terms of PRR even for nodes far from the sink as it avoids bad links. The authors also state that ETX balances the load better among the nodes.
The authors also investigate the route prevalence ("average ratio of time during which the same route has been observed"). It shows that OF0 is quite stable while for ETX and LQI the routes change more often. The observation that ETX has a higher rate of parent changes than OF0 has also been discovered in the experiments shown in Table 3.1.
Oana Iova et al. conclude that no single (existing) OF yields the best results. There are indeed always trade-offs between a higher latency (LQI), higher instability (ETX) or lower PRR (OF0). This is a good example showing that the design choice in RPL of leaving the OF implementation up to the application designer is beneficial. The choice of OF indeed is highly depended on the WSNs goals and requirements.

In their paper *Energy-Efficient Low Power Listening for Wireless Sensor Networks in Noisy Environments* [29] Mo Sha et al. investigate the benefits of an adaptive CCA control algorithm "under normal channel conditions and controlled 802.11n traffic". The main goal of their Adaptive Energy Detection Protocol (AEDP) is to mitigate false wakeups of the radio / MAC while it is in low power listening mode. In this thesis, the focus to automatically change the CCA lays mostly to allow sending packets even under interference, while also reducing the energy impact if a high level of interference is present.
The measurements of Mo Sha et al. were performed using Tmote Sky nodes, but running on TinyOS and its BoX-MAC-2 MAC layer. According to the authors Contiki OS' default RDC `ContikiMAC` already provides some mitigations to false wakeups due to performing two CCA checks shortly after each other to rule out short (false) wakeups. Longer interference burst as created in this thesis (`JL_WIFI4`, `JL_CONTN`) still affect `ContikiMAC`.
The authors use different values for the CCA threshold: one used for low power listening (*wakeup threshold*) and the other one used for collision avoidance. Only the wakeup threshold is changed and the threshold used for transmission is kept at the same level but the authors state that AEDP could also be applied when transmitting.
The experiment are carried out on 20 nodes at varying distances from a 802.11n sender and receiver show that changing the CCA threshold has the ability to reduce the number of false wakeups from more than 97.8 % to 0 %. The higher wakeup threshold will result in nodes not waking up even for valid messages. Mo Sha et al. observe that using a fixed more aggressive wakeup threshold is not of good use, as the interference highly changes over time, application and distance from the interference source.

The AEDP implementation uses three runtime variables to estimate a new wakeup threshold. ETX is used over a sliding window but it is estimated using a sequence number in the incoming packets instead of after sending a packet as in RPL. There the number of required transmissions until an ACK has been received is used. As a further variable the authors calculate the wakeup rate over the same sliding window and the wakeup rate over whole application lifetime.
For each of the three variables a threshold is defined and the algorithm tries to find the minimum wakeup threshold so that each variable stays below the defined threshold. tTe newly estimated wakeup threshold is limited between a minimum and maximum value. The minimum value is set to the noise floor at compile time and the maximum value is determine by the minimum RSSI of the incoming packets. As the RSSI of the packets changes over time or new nodes with a lower RSSI at the receiver might be added, AEDP sets the wakeup threshold periodically to the noise floor value. This ensures that new packets and thus RSSI values are received and the wakeup threshold is limited accordingly. In contrast to this approach, this thesis periodically measures the noise floor independently of the received packets and so no periodic probing using a lower wakeup threshold is required.
The approach of Mo Sha et al. reduces the number of false wake ups and thus the duty cycle by about 40 % in their tested setup. They also performed experiments using CTP and 55 nodes. There AEDP reduced the duty cycle by 35 % while also reducing the end-to-end ETX by 11 %. This setup has been run on an overlapping IEEE 802.11 channel.
One limitation of AEDP is that it is ineffective for links with an RSSI close to or below the noise floor. The wakeup threshold will be set to the minimum RSSI and thus AEDP cannot reduce the false wakeups due to noise. The approach shown in Section 4.2 does not suffer from this limitation, as the wakeup threshold is always set to a value ϵ above the current noise floor.

Positive results of changing the CCA threshold has also been shown by Boano et al. while *mitigating the Adverse Effects of Temperature on Low-Power Wireless Protocols* [4]. The focus in this paper is to change the CCA threshold based on the temperature variations in the network. Increasing or decreasing temperature has a bad influence on the performance of radio transceiver and thus the network connectivity.
The authors show experimentally that an increased on-board temperature erroneously lowers the measured RSSI, tricking the sender to send a message even tough the radio channel is not free, resulting in a packet collision and wasted energy. On the receiver side, due to the lowered RSSI of an incoming packet the radio is not kept awake to receive the packet.
As a result of Boano et al.'s experiments, they conclude that a link may "indeed still offer good performance if the CCA threshold is dynamically adapted to the on-board temperature variations".
By implementing an adaption algorithm to change the nodes CCA threshold based on the measured nodes on-board temperature, the authors achieve a 87 % higher packet reception ratio. The algorithm needs a model for the relationship between signal strength attenuation and the temperature. The authors found this to be a linear factor, but depending both on the sending and receiving nodes temperature. To retrieve the neighbour nodes temperature, they piggypack the temperature information onto RPL's beacons.
This paper, again, makes it clear that a dynamic CCA threshold has a high potential to improve the networks reliability and dependability.

Chapter 7

Conclusions and Future Work

RPL is the emerging standard for routing IPv6 packets in a WSN. It has been standardized by the IETF and is now widely investigated in research.
Some WSN applications require a high reliability and dependability. To analyse the suitability of RPL for those applications, the first part of this thesis investigates RPL's performance while the network is subjected to radio interference.
Towards this goal, tools and scripts are created to instrument, analyse, and evaluate Contiki OS' RPL in a network of about 30 nodes in a real testbed. A mixture of Python and Bash scripts allow for a semi automatic work-flow from compiling the changes, deploying the firmware to the nodes and collecting the requested data. The data is then processed using Python scripts to generate tables, plots and statistics as required.
The results of a longer experimental campaign show that RPL's performance highly depends on the underlying link-layer protocol. `ContikiMAC` is indeed easily blocked by a strong enough jammer, which leads to a low PRR and a high energy waste, reducing the nodes battery lifetime significantly.
`ContikiMAC`'s CCA threshold is found to be an adjustable knob to improve the network performance under heavy interference.
Hence, in the second part of this thesis, an adaptive CCA threshold algorithm is proposed and implemented. The implemented AutoCCA algorithm is then evaluated. The most significant experiments from the experimental campaign are repeated with the proposed algorithm enabled, and show that dynamically changing the CCA threshold increases the PRR from 9 % to about 60 % in the case an jammer generating a continuous carrier, while also reducing the energy consumption 69 %.
As far as the experiments go, no big drawbacks of the suggested algorithm have been found. The added code size is with about 150 lines quite small. The additional energy consumption added by the CPU and radio-on time (50 ms every 10 s) is largely compensated by an overall energy reduction.

Future work on this topic might be implementing the β parameter of Equation 4.3. This would integrate data collected from RPL such as the number of available neighbours into the CCA threshold adaption algorithm to enforce a smaller set of (better) parents. This could also be used to further reduce the energy consumption if no noise is present, as due to a higher CCA threshold the number of false wake ups would be reduced.
The AEDP algorithm proposed by Mo Sha et al. [29] could also be implemented for Contiki

OS and tested using RPL.
As the performance of RPL highly depends on the underlying link-layer protocol, trying other link-layer protocols such as X-MAC or investigating other improvements to the link-layer such as forward error correction could be done. Evaluating using the RSSI or LQI as a metric in an objective function would also be an interesting topic. This would extend the works of Oana Iova et al. [19] made in simulation to testing it in a testbed.
Instead of using JamLab to generate the interference a setup using real IEEE 802.11 devices and their introduced interference could be installed and evaluated in a similar approach.

Bibliography

[1] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: a high-fidelity sensing testbed. *IEEE Internet Computing*, 10(2):35–47, March 2006.

[2] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 43–56, New York, NY, USA, 2008. ACM.

[3] C. A. Boano, T. Voigt, C. Noda, K. Römer, and M. Zúñiga. JamLab: Augmenting sensornet testbeds with realistic and controlled interference generation. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 175–186, April 2011.

[4] C.A. Boano, K. Romer, and N. Tsiftes. Mitigating the adverse effects of temperature on low-power wireless protocols. In *Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on*, pages 336–344, Oct 2014.

[5] Carlo Alberto Boano, Marco Zúñiga, James Brown, Utz Roedig, Chamath Keppitiyagama, and Kay Römer. TempLab: A Testbed Infrastructure to Study the Impact of Temperature on Wireless Sensor Networks. In *Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, IPSN '14, pages 95–106, Piscataway, NJ, USA, 2014. IEEE Press.

[6] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. 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, pages 307–320, New York, NY, USA, 2006. ACM.

[7] Geoff Coulson, Barry Porter, Ioannis Chatzigiannakis, Christos Koninis, Stefan Fischer, Dennis Pfisterer, Daniel Bimschas, Torsten Braun, Philipp Hurni, Markus Anwander, Gerald Wagenknecht, Sándor P. Fekete, Alexander Kröller, and Tobias Baumgartner. Flexible experimentation in wireless sensor networks. *Commun. ACM*, 55(1):82–90, January 2012.

[8] Manjunath Doddavenkatappa, Mun Choon Chan, and A. L. Ananda. *Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed*, pages 302–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] T. Ducrocq, J. Vandaële, N. Mitton, and D. Simplot-Ryl. Large scale geolocalization and routing experimentation with the SensLAB testbed. In *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2010)*, pages 751–753, Nov 2010.

[10] Adam Dunkels. SICSlowpan - Internet-connectivity for Low-power Radio Systems. December 2009.

[11] Adam Dunkels. The ContikiMAC radio duty cycling protocol. *SICS Technical Report T2011:13 ISSN 1100-3154*, December 2011.

[12] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[13] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14, New York, NY, USA, 2009. ACM.

[14] E. Guizzo. Into deep ice [ice monitoring]. *IEEE Spectrum*, 42(12):28–35, Dec 2005.

[15] Dong Han and O. Gnawali. Performance of RPL under wireless interference. *Communications Magazine, IEEE*, 51(12):137–143, December 2013.

[16] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2Nd International Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality*, REALMAN '06, pages 63–70, New York, NY, USA, 2006. ACM.

[17] J. Horneber and A. Hergenröder. A survey on testbeds and experimentation environments for wireless sensor networks. *IEEE Communications Surveys Tutorials*, 16(4):1820–1838, Fourthquarter 2014.

[18] W. Hu, Van Nghia Tran, N. Bulusu, C. T. Chou, S. Jha, and A. Taylor. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 503–508, April 2005.

[19] O. Iova, F. Theoleyre, and T. Noel. Stability and efficiency of RPL under realistic conditions in wireless sensor networks. In *2013 IEEE 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 2098–2102, Sept 2013.

[20] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for “smart dust”. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 271–278, New York, NY, USA, 1999. ACM.

[21] A. Lavric, V. Popa, and S. Sfichi. Street lighting control system based on large-scale WSN: A step towards a smart city. In *Electrical and Power Engineering (EPE), 2014 International Conference and Exposition on*, pages 673–676, Oct 2014.

[22] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Information Processing in Sensor Networks (IPSN), 2013 ACM/IEEE International Conference on*, pages 153–165, April 2013.

[23] Nguyen Thanh Long, N. De Caro, W. Colitti, A. Touhafi, and K. Steenhaut. Comparative performance study of RPL in wireless sensor networks. In *Communications and Vehicular Technology in the Benelux (SCVT), 2012 IEEE 19th Symposium on*, pages 1–6, Nov 2012.

[24] Moteiv Corporation. Tmote Sky: Datasheet. http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf, Jun 2006.

[25] A. Mpitziopoulos, D. Gavalas, C. Konstantopoulos, and G. Pantziou. A survey on jamming attacks and countermeasures in WSNs. *IEEE Communications Surveys Tutorials*, 11(4):42–56, Fourth 2009.

[26] Răzvan Musăloiu-E. and Andreas Terzis. Minimising the effect of WiFi interference in 802.15.4 wireless sensor networks. *Int. J. Sensor Networks*, 3(1):43–54, December 2008.

[27] F. Pu, C. Li, T. Gao, J. Pan, and J. Li. Design and implementation of a wireless sensor network for health monitoring. In *Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on*, pages 1–4, June 2010.

[28] Divya Sakamuri. NetEye: A wireless sensor network testbed. Master's thesis, Graduate School of Wayne State University, Detroit, MI, USA, 2008.

[29] Mo Sha, Gregory Hackmann, and Chenyang Lu. Energy-efficient low power listening for wireless sensor networks in noisy environments. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN '13, pages 277–288, New York, NY, USA, 2013. ACM.

[30] Texas Instruments Inc. MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller (Rev. G). http://www.ti.com/lit/ds/symlink/msp430f1611.pdf, 2011.

[31] Texas Instruments Inc. CC2420: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. http://www.ti.com/lit/ds/symlink/cc2420.pdf, 2013.

[32] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[33] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: a wireless sensor network testbed. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 483–488, April 2005.

[34] H. Xiao and H. Ogai. A distributed localized decision self-health monitoring system in wsn developed for bridge diagnoses. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 23–28, May 2011.

[35] J. Yang, J. Portilla, and T. Riesgo. Smart parking service based on wireless sensor networks. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 6029–6034, Oct 2012.