Implementing and Evaluating Communication Protocols for Reliable Multicast in Wireless Sensor Networks

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sandro Beffa
2012

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

List of Figures

Code Listings

List of Tables

Chapter 1

Introduction

1.1 Motivation

Tiny and cheap embedded systems equipped with a radio interface observing a border region is a practical application of a so-called wireless sensor network (WSN). With decreasing size and cost of integrated circuits (ICs), the number of possible applications for WSNs is increasing every day.

Wireless sensor networks are of great and emerging interest to today's research. They can be characterized as a network built of tiny and cheap embedded systems, which have very limited resources available [1]. These embedded systems are referred to as sensor nodes (SNs) and are acting autonomously: They are deployed without an external power supply and are equipped only with an on-board battery. Energy is therefore a very scarce resource. But also the computational power of the sensor nodes is very limited. Additionally to the necessary radio device, the sensor nodes are equipped with different sensors to accomplish their tasks.

There is a great number of possible applications of a WSN, reaching from military to civil usage. With increasing size of a WSN, managing and monitoring each individual sensor node during its lifetime becomes more difficult, especially when different sensor nodes platforms are used. To simplify the management of heterogeneous WSNs, MARWIS (Management Architecture for Wireless Sensor Networks) [2] was developed at the University of Bern.

MARWIS introduces the usage of a wireless mesh network (WMN) as a backbone for building a heterogeneous WSN. Other than a WSN, a WMN is built of more powerful hardware. The nodes are typically equipped with more powerful central processing units (CPUs) and radio devices. Additionally these wireless mesh nodes also usually support wired networking. MARWIS features the ability to monitor, configure and update every individual sensor node. The possibility to update a sensor node over the air is of great importance to simplify the maintenance of a WSN. The update mechanism of MARWIS is capable of updating specific application parts of individual sensor nodes without having to replace the whole operating system.

The architecture of MARWIS consists of three components:

- management station
- WSN manager
- SN agent

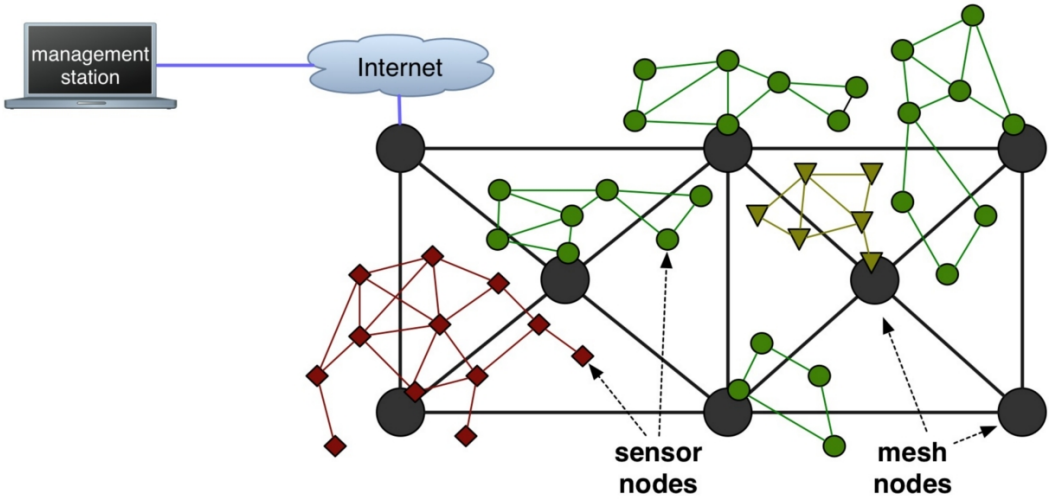

Figure 1.1: MARWIS architecture [2]

Figure 1.1 shows a heterogeneous WSN scenario using WMNs as backbones to different WSN subnets. The management station is a common desktop PC connected to the Internet. The black circles symbolize the wireless mesh nodes, which are meshed to each other. The black circle is the mesh gateway and is directly connected to the Internet. The WSN manager runs on the mesh nodes and manages specific sensor nodes. The green, red and yellow shapes are the actual sensor nodes, on which the SN agent runs. This SN agent is responsible for executing the specific management tasks on the sensor node. Each shape represents a different sensor node platform, meaning that the sensor nodes are built of different CPU architectures and radio devices. The MARWIS architecture introduces a segmentation of the whole heterogeneous WSN into various smaller homogeneous WSN subnets.

Each sensor node supports TCP/IP and is therefore reachable from the Internet. One specific sensor node is used as a gateway to the WSN subnet and is directly connected to the wireless mesh node. The mesh node establishes a SLIP [1] connection to the sensor node and forwards data addressed to this WSN subnet to the attached sensor node.
To support code updates in one specific WSN subnet, there must be the possibility to distribute the code update fast and energy efficient in the WSN. The Sensor Node Overlay Multicast protocol (SNOMC) [3] is a overlay multicast protocol designed to efficiently distribute data in a

[1] "Serial Line Internet Protocol" connections encapsulate a common IP packet and add a special "SLIP END" character. Data is transmitted via a RS232 connection to the attached destination. There the "SLIP END" character is removed and the IP packet is forwarded as usual.

WSN. The main goal of SNOMC is to avoid redundant unicast connections, thus minimizing the energy consumption of the update process. To accomplish this, the user datagram protocol (UDP) was chosen as transport protocol, because of the slight overhead UDP offers. Since UDP connections are stateless and not reliable, SNOMC comes with its own reliability mechanism based on NACKs and closing positive ACKs. Three different caching strategies are supported:

- only on the source node
- on the branching nodes
- on every intermediate node

SNOMC was implemented on top of the Contiki OS [4]. To evaluate SNOMC in a real world scenario, SNOMC has to be compared to other common communication protocols used in WSNs, for example Directed Diffusion (DD) [5], Multipoint Relay (MPR) [6] or Flooding. Since neither of these protocols are available for the Contiki OS, they needed to be implemented. That leads to the motivation of this thesis: Three common communication protocols are implemented on top of the Contiki OS and evaluated, so SNOMC can be compared to the data gathered in a real world scenario.

1.2 Scenario

The main purpose of the implemented communication protocols is to distribute code updates in a WSN. The actual size of the code update was limited to 1000 bytes, since Contiki OS is able of updating only the application part of a sensor node. In contrast to common traffic found in WSNs, the traffic used to transport binary data for a code update is bursty: There is no sparse and continuous traffic flow, instead there is short and very intensive traffic. Three common communication protocols are implemented on top of Contiki OS:

- Flooding
- Multipoint Relay
- Directed Diffusion

Contiki is equipped with a very small TCP/IP stack called µIP, which brings TCP/IP networking to the world of embedded systems. As the case with SNOMC, we chose UDP as transport protocol on top of IP and implemented the communication protocols on the application layer. The same NACK-based reliability mechanism used in SNOMC is implemented to reduce WSN traffic. Similar to SNOMC, the implemented communication protocols support three caching strategies:

- no caching
- every intermediate node caches data
- proactive caching: the intermediate nodes are actively requesting lost data

To get real world data, the implemented communication protocols need to be evaluated in a real world WSN. Mentioned communication protocols were therefore evaluated in a real WSN testbed using TARWIS [7], which is a testbed management architecture for WSNs located and developed at the University of Bern. Since the size of the maximum transmission unit (MTU) of the most radio transmitters found in sensor nodes is smaller than the size of the chosen payloads, these payloads have to be split into various fragments. Two different payload sizes were evaluated: 70 bytes, which is equal to one UDP packet, and 1000 bytes, which is equal to 15 UDP packets depending on the chosen protocol design.

Energy-efficiency is crucial for a real world communication protocol used in a WSN, because of the limited energy resources of the sensor nodes. Since radio communication consumes over 90% of the total energy, the only way to save energy is to disable the radio device for short periods. Special energy optimized media access layer (MAC) protocols take care of this.

To evaluate the impact of these energy optimized MAC protocols, the communication protocols were evaluated with two different MAC protocols:

- NullMAC
- ContikiMAC

1.3 Goal

The goal of this Bachelor thesis is to implement the three mentioned communication protocols on top of the Contiki OS, running on TmoteSky sensor nodes. The implemented communication protocols shall be evaluated in a real world WSN, located at the University of Bern. As evaluation metrics shall be used:

- the time to transmit data to all receivers
- the number of sent frames and the number of collisions on the MAC layer
- the used energy for transmitting data to all receivers

Along with these evaluation metrics, the implemented communication protocols shall be evaluated with the MAC protocols mentioned above.

1.4 Structure of the Thesis

This bachelor thesis is structured as follows: The second chapter introduces important related work, especially TARWIS and the Contiki OS. The third chapter describes the design of the implemented communication protocols in detail: the protocol stack, the reliability mechanism and the sequence diagrams of the communication protocols. The fourth chapter focuses on technical aspects: The actual implementation of the communication protocols is described here. The fifth chapter is dedicated to the evaluation and the discussion of the obtained results. Finally, the sixth chapter concludes the work done and discusses future work.

Chapter 2

Related Work

2.1 TARWIS

The Testbed Management Architecture for Wireless Sensor Network Testbeds (TARWIS) [7] is a testbed management solution, which simplifies the setup of WSN-based experiments: Equipped with a web-based graphical user interface (GUI), TARWIS allows the user to upload software images for the used sensor nodes and to schedule experiments. TARWIS takes care of flashing the software image to the chosen sensor nodes and of gathering the output of these sensor nodes.

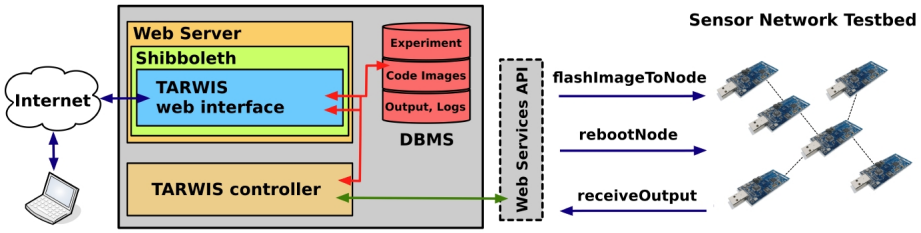

Figure 2.1: TARWIS architecture [7]

Figure 2.1 illustrates the system architecture of TARWIS. The main component is a portal server, on which the TARWIS controller, a database management system (DBMS) and the TARWIS web interface runs. The TARWIS controller is responsible of executing actions on the managed sensor nodes. To communicate with these nodes, a common web services API is used, which has to be implemented by the used sensor nodes. The whole communication between the TARWIS controller and the sensor nodes is therefore common Hypertext Transfer Protocol (HTTP) traffic, consisting of exchanging HTTP POST and GET messages. The DBMS is used to store the retrieved experiment data, the software images and additional output and log data.

2.2 TmoteSky Sensor Nodes

TmoteSky is the name of an ultra low power sensor node platform, which comes in a very compact design [8]. It is powered by an `msp430` microcontroller unit (MCU) from Texas In-

struments, which runs at 8 MHz. For wireless communication, the TmoteSky platform comes with the CC2420 radio transceiver from Chipcon. Additionally, multiple sensors are included for environmental surveillance like humidity, light and temperature. The msp430 is a well known 16-Bit Reduced Instruction Set Computer (RISC) low power MCU, which supports five power save modes. It draws 0.2 µA in the lowest power save mode and is equipped with 10 KB of main memory and 1 MB of secondary storage to store the sampled sensor data. The CC2420 is a single-chip 2.4 GHz IEEE 802.15.4 compliant RF transceiver designed for low power and low voltage wireless applications [9]. It is a packet-oriented radio, which transmits 250 kbps at the maximum. The output power of the CC2420 is programmable and reaches from -24 dBm to 0 dBm, which allows a 50m range indoors and 125m outdoors based on the onboard antenna of the TmoteSky board. As the license free 2.4 GHz industrial, scientific and medical (ISM) radio band is also used by other applications like WLAN or Bluetooth, the low output power of the CC2420 may be problematic. If all those applications have to share one transmission medium, it is very likely that radio communication of the weakest is disturbed, although all applications formally use different frequency bands. Especially Bluetooth has the reputation of a well known radio interferer through its frequency-hopping spread spectrum (FHSS) method. With this technique, the radio transceiver uses a lot of different frequency channels to transmit the radio signal, and hops rapidly from channel to channel. This technique obviously claims a lot space in the ISM radio band and therefore may influence the radio communication of weaker transceivers like the CC2420.

Figure 2.2 shows a TmoteSky sensor node. This node comes in a very compact form of 6.5cm height, 3cm width and 2cm depth. Equipped with an universal serial bus (USB) port and an onboard bootloader, this sensor node is flashable without any additional device. This advantage makes it easy-to-use and very user friendly.

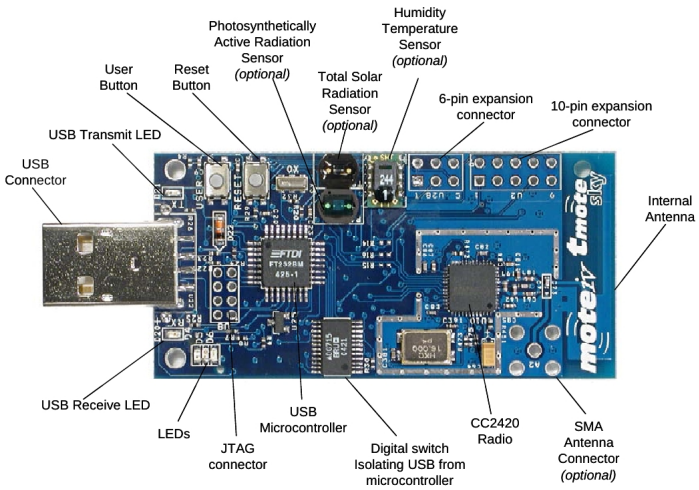

Figure 2.2: Front side of the TmoteSky sensor node [8]

2.3 Contiki Operating System

Contiki is a lightweight and flexible operating system for sensor nodes, which was developed at the Swedish Institute of Computer Science (SICS) [10]. Contiki differs from other popular operating systems for sensor nodes through the possibility of replacing certain parts of the operating system at run-time over the network. Through this over-the-air (OTA) update mechanism it is not necessary to replace the whole firmware image for deploying a small bug fix or to add new applications. It is possible to replace only certain processes or services running on the sensor node. This leads to significant energy savings, since the energy used to transmit a few hundred bytes for a bug fix is much less than the energy used to transmit the whole firmware image. Besides this big advantage, Contiki introduces a powerful and easy-to-use event-driven scheduler, which simplifies the creation of event-driven code. To communicate with other sensor nodes or with the Internet, Contiki comes with the worlds smallest TCP/IP stack called µIP [11]. Contiki is highly portable and written in C. Popular supported platforms are the `TmoteSky` from Texas Instruments and the `AVR Raven` from Atmel. The Contiki kernel is event-driven and supports a simple first in / first out (FIFO) scheduler. Preemptive multitasking can be added to specific processes by using an application library.

2.3.1 Contiki Processes and Scheduling

Event-driven programming is the favored programming style for developing applications for embedded systems, since this approach can keep the memory overhead down. The memory saving results from the omission of processes and their corresponding data structures like the process control block (PCB) and the stack, which can be found in any multitasking capable operating system. The abstraction of a process gives the developer the possibility to execute simultaneously different tasks on only one central processing unit (CPU). The operating system takes care of switching between the different tasks. This switching is called "context switch", and consists of saving and restoring the state of a process, which is very time consuming, since a lot of data has to be copied. As opposed to using processes, a PCB and a stack are not necessary by using event-driven programming. This programming style consists of describing the application as a finite state machine, where only the different states have to be saved, and not the whole stack and a PCB. Depending on a certain state, a specific action is executed. Implementing an application as a finite state machine is often difficult and error prone [12].

Contiki supports some sort of lightweight processes, which are a mix of ordinary processes and event-driven programming called *protothreads* [12]. Protothreads combine the benefits of following two approaches: the low memory footprint of event-driven programming, since protothreads does not have its own stack, and a process-like programming style with a blocking wait statement. This new approach simplifies the development of event-driven code, since the application can be described as a linear sequence of program statements. The blocking wait statement is triggered by the `PT_WAIT_UNTIL()` function. This statement blocks the protothread: the code execution is stopped and the next protothread in the process queue is executed. This blocking wait statement works together with the FIFO scheduler of Contiki. Without this blocking wait statement, only one protothread would be executed, thus monopolizing the CPU.

This situation results from the fact, that a FIFO scheduler requires each process to run to its

completion, before the next process can be scheduled. A protothread consists of an infinite main loop, thus implying the need of a mechanism, that allows the protothread to wait for a certain event without monopolizing the CPU by using busy waiting. The Contiki scheduler and the protothreads work together by moving the decision whether and when a protothread should give away the control of the CPU to the developer. To achieve a multitasking-like executing flow, a developer has to block his protothread as often as possible by using the `PT_WAIT_UNTIL()` statement to share the CPU as much as possible with other protothreads. An important side-effect of the omission of an own stack for a protothread is the fact, that all used variables have to be declared global and *not* local in the scope of the protothread. A protothread also is referred to as a Contiki process.

2.3.2 The µIP Stack

µIP is a lightweight TCP/IP-Stack, which was developed to run on very resource constraint 8-bit architectures. Despite its small memory footprint, µIP fulfills the subset of RFC1122 needed for full host-to-host interoperability [11]. To use as little memory as possible, µIP uses one global packet buffer, that holds incoming and outgoing packets. This buffer can store only one packet. Therefore, the packet buffering has to be implemented either by the MAC protocol or the application using µIP. The latter approach is used in this thesis, as discussed in chapter 4.3.

Using TCP/IP communication on 8-bit architectures has its challenges, as the transmission control protocol (TCP) and the user datagram protocol (UDP) use 32-bit checksums in their protocol headers. Calculating a 32-bit checksum on an 8-bit architecture takes many more clock cycles than on an 32-bit architecture, since the 32-bit checksum has to be stored in the main memory and cannot be processed directly in the 8-bit wide CPU registers. The challenge lies in an optimized implementation of this 32-bit checksum algorithm on an 8-bit architecture. µIP solves this challenge by separating its functionality into two parts: one generic part, which is independent of the used architecture, and an architecture specific part (like calculating checksums). Therefore, only a small specific part of µIP has to be adapted to the used architecture.

µIP was implemented as a common protothread on top of Contiki, and therefore uses events to signal the arrival of new packets. To use µIP, the application protothread has to open a new connection to the desired IP address. µIP then stores the process ID (PID) of the application protothread to the connection state data structure. Every time a new packet arrives, µIP posts a `tcpip_event` to the registered protothread, which is unblocked by the scheduler as soon as it is its turn. Listing 2.1 illustrates the opening of a UDP unicast connection to the port 2500 of the host with the IP address `192.168.1.20` and the transmission of one UDP packet with the payload “HELLO”:

Listing 2.1: Sending an UDP packet

```
1 static struct uip_udp_conn *udp_conn;
2 static uip_ipaddr_t ipaddr;
3 static char* msg="HELLO";
4 uip_ipaddr(&ipaddr, 192,168,1,20);
5
6 udp_conn = udp_new(&ipaddr, UIP_HTONS(2500), NULL);
7 uip_udp_packet_send(udp_conn, &msg, sizeof(msg));
```

The data structure `uip_udp_conn` stores the PID of the application protothread, to which the `tcpip_event` is posted. Additionally, the destination port and destination IP address is stored by this data structure. The µIP function `udp_new` opens the UDP unicast connection and the function `uip_udp_packet_send` triggers the the transmission of the data found at the supplied memory address *&msg*. Listing 2.2 illustrates the code on the receiver part for listening for a `tcpip_event` and processing the arrived packet:

Listing 2.2: Receiving an UDP packet

```
1 while(1){
2   PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event)
3
4   if (uip_newdata()){
5     uint8_t len = uip_datalen();
6     void   *ptr = uip_appdata;
7   }
8 }
```

The statement `PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event)` blocks this protothread until a `tcpip_event` arrives. The preprocessor macro `uip_newdata()` checks if a new packet arrived, and the `uip_datalen()` function returns the size of the arrived data in bytes. `uip_appdata` is a void pointer to the packet payload.

2.3.3 RIME

RIME is a lightweight layered communication stack for WSNs, which is part of the Contiki OS [13]. The RIME stack differs from traditional network stacks in such a way, that each of its layers are very thin. Each layer therefore introduces very limited functionality and accomplishes only one certain task. This approach reduces the implementation complexity and leads to a very easy-to-understand design. The RIME stack offers methods for unicast and broadcast communication, that are optimized for the needs of embedded systems. RIME uses very short headers from one to the next layer and avoids resource intensive 32-bit checksums. The Rime stack can be used as a standalone communication stack without µIP. This possibility reduces the communication overhead, but brings the disadvantage of isolating the WSN from common Internet communication. This drawback can be solved through the usage of a proxy service, that translates TCP/IP packets to RIME packets and vice versa. Another possibility is to use µIP on top of RIME. This is the default configuration of Contiki. IP packets are tunneled over the RIME stack, which results in an additional 2 byte header for each IP packet. This approach

has its strength in its flexibility: To communicate with the Internet, the common TCP/IP-style communication offered by µIP can be used. The data supplied by the WSN can be accessed from any usual desktop PC or mobile device connected to the Internet. On the other side, the lightweight communication offered by the RIME stack can be used for management tasks, that only take place inside the WSN. Neighborhood discovery could be such a management task.

2.3.4 MAC Protocols

A media access control (MAC) protocol is mainly responsible for sharing the transmission medium between multiple clients, which want to use this transmission medium at the same time. Besides that task, possible error correction of the data received from the physical layer and the radio duty cycling (RDC) falls within the area of competence of a MAC protocol. To save energy, the radio transceiver has to be turned off as much as possible: A TmoteSky sensor node draws 21.8 mA, while the `msp430` MCU is on and the `CC2420` radio transceiver is in listening mode. With the radio turned off, the current consumption drops to 1.8 mA, which is about 8.25 % of the whole energy consumption. In other words, 91.75 % of the energy consumption is used for the radio communication [8]. Therefore, energy optimized MAC protocols, which turn of the radio for short periods, are crucial to the lifetime a WSN. Contiki splits the traditional MAC protocol into two smaller, task-specific protocols:

- a MAC protocol: responsible for accessing the transmission medium using CSMA/CA
- a RDC protocol: responsible for power saving.

In terms of Contiki, the MAC protocol is responsible of checking the transmission medium whether someone else is transmitting data or not. If the transmission medium is free, the data can be transmitted. If the transmission medium is occupied, the MAC protocol has to wait for a certain time and then to try again to send the data. This mechanism is called Carrier Sense Multiple Access with collision avoidance (CSMA/CA). The other part of the MAC protocol, handling the radio duty cycling, is called RDC protocol. The RDC protocol takes care of the energy saving mechanisms, which involve turning off the radio transmitter as much as possible without decreasing the transmission performance too much.

This distinction of a common MAC protocol into two parts can sometimes be very confusing. The following names, `NullMAC` and `ContikiMAC`, describe common MAC protocols for Contiki. That means they consist of the two parts explained above: a MAC and a RDC part. Therefore the MAC part of `NullMAC` is called *nullmac*, and the RDC part is called *nullrdc*. For `ContikiMAC`, the MAC part is called *csma*, and the RDC part is called *contikimac*. This separation shall be denoted by writing the names for common MAC protocols in `CAPITALS`, and the names for a specific part of a MAC protocol in *lower case*.

`NullMAC`

`NullMAC` is the simplest possible MAC protocol, which is supported by Contiki. Both parts of `NullMAC` actually do nothing, therefore the name *`Null`*`MAC`. That means, *nullmac* does only check, if the transmission medium is free to send the packet. If the transmission medium is

occupied, the packets are dropped without any attempt to resend. Analogously, *nullrdc* never turns off the radio transmitter. The packets are transmitted with the maximum performance, but also with the maximum energy consumption.

ContikiMAC

`ContikiMAC` is an energy optimized MAC protocol, and is the default MAC protocol of Contiki since release 2.5 [14]. The energy saving mechanisms used in `ContikiMAC` are inspired by other popular MAC protocols, and consist of following three measures:

- periodic wake-ups, inspired by B-MAC [15], X-MAC [16] and Box-MAC [17]
- wake-up strobes, inspired by Box-MAC [17]
- phase-lock optimization, inspired by WiseMAC [18]

`ContikiMAC` unifies these three energy saving measures to one RDC protocol called *contikimac*. These unified mechanisms work as follows: First, each `ContikiMAC`-enabled sensor node turns on the radio transmitter periodically only for a very short time. In this short time period, *contikimac* checks for radio activity using an inexpensive Clear Channel Assessment (CCA), which uses the Received Signal Strength Indicator (RSSI) to give an indication for radio activity on a certain channel. A packet transmission is detected by comparing the RSSI value with a given threshold. If the RSSI value is higher than the threshold, the CCA is positive and *contikimac* assumes that a packet transmission is on the way. This mechanism is called *periodic wake-ups*.

If a packet transmission is detected, the sensor node keeps the radio transmitter on until the whole packet has been received. To signal the successful reception of the packet, the receiver sends an acknowledgment and turns off the radio. This is only working, if the sender sensor node continuously resends the packet, until it receives an acknowledgment. Those periodically resent packets are called *wake-up strobes*. If the sender wants to transmit a broadcast packet, it has to be resent for the full wake-up interval, because every node has to be reached and every node only wakes up once in the entire wake-up interval.

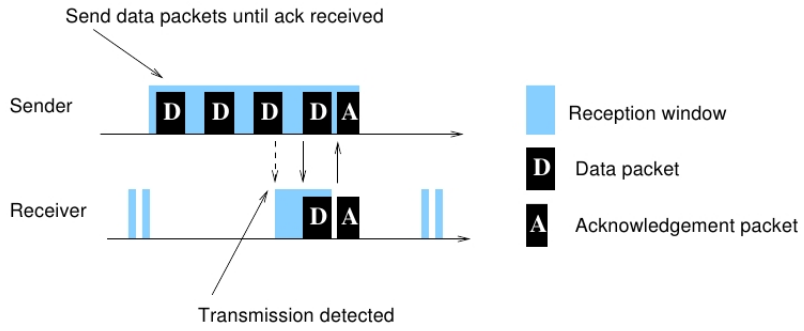

Figure 2.3: Unicast transmission using `ContikiMAC` [14]

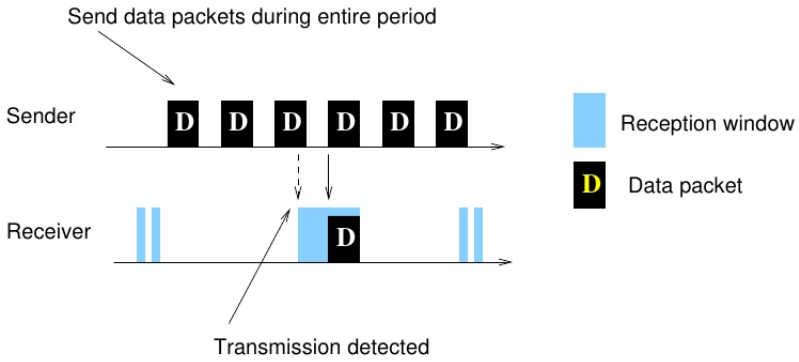

Figure 2.4: Broadcast transmission using `ContikiMAC` [14]

Figure 2.3 and 2.4 illustrate these two situations. The main difference between a unicast and a broadcast transmission using `ContikiMAC` is the long rebroadcast time for a broadcast transmission, since the sender has to make sure to reach all desired receivers. This drawback is really serious for broadcast based communication protocols like Flooding or MPR, because it slows down the data transmission enormously, as described in chapter 5.5. The third mechanism to optimize the energy consumption is called *phase-lock optimization*. Using this technique, the sender tries to learn when a certain receiver wakes up. Since the wake-up of a certain receiver is periodic, the sender can estimate the next wake-up of the receiver after the first successful packet transmission. With this estimated sleep interval, the sender is able to time the first transmission of a wake-up strobe more accurately.

2.3.5 Energy Measurement

To make a statement about the energy used to transmit a certain amount of data, we need a procedure to measure the energy consumption of all participating sensor nodes. Using a hardware-based measurement method, this task can be very costly with increasing size of the WSN. A simple hardware-based energy measurement consists of attaching an oscilloscope in series with the power source of a sensor node to measure the current potential and to calculate the current draw. Doing this for the whole WSN would be very expensive and not practicable. On the other hand, software-based energy measurement methods are inexpensive and yield accurate results as pointed out by [19]. Powertrace [19] is a software-based energy measurement tool developed on top of Contiki and is implemented as a common protothread, which can be started as soon as the energy measurement begins. To measure the used energy, powertrace uses so-called *energest* values [20]. These *energest* values are indeed only clock ticks of the used MCU, and can be converted to seconds by dividing them by the clock rate of the used MCU. In case of a TmoteSky sensor node, which uses a `msp430` MCU, one second is equal to 32768 clock ticks. To give a meaning to these *energest* values, powertrace needs instrumentation of the used device drivers and RDC protocols. With this instrumentation, powertrace is able to provide *energest* values for interesting energy consumers like the CPU or the radio transmitter. To get the *energest* value for the radio communication, powertrace simply sums up the *energest* values for the listening and transmitting of packets during the desired time period. These *energest* values describe how

much time in clock ticks was spent for radio communication or common tasks using the CPU.

Based on those *energest* values, the energy spent can be calculated by converting these values to seconds and multiplying them with the used power in Watt. The current consumption for certain actions like listening for data or transmitting data can be found in the datasheet of the used component [8]. For the `CC2420`, used by the TmoteSky sensor node, the current consumption for listening is 21.8 mA, and 19.5 mA for transmitting. To convert these values to Watt, they need to be multiplied by the current potential, which is 3.6 V in case of a TmoteSky sensor node. That amounts to a power consumption of 78.48 mW for listening, and 70.2 mW for receiving. Multiplying seconds with Watt, which is $\frac{Joule}{Second}$, gives the energy consumption expressed in Joule.

Chapter 3

Design of Communication Protocols for Reliable Multicast

3.1 Protocol Stack

The protocols implemented in this thesis are application layer protocols, since they are running on top of the user datagram protocol (UDP). Implementing a communication protocol on the application layer simplifies the implementation, but generates an increased overhead. The additional overhead originates from the fact that each protocol message always has to travel up and down the *whole* protocol stack.

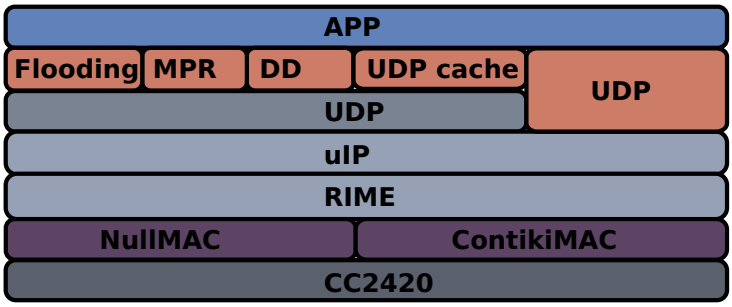

Figure 3.1: Used protocol stack

Figure 3.1 shows the protocol stack used to implement Flooding, Multipoint Relay and Directed Diffusion. On the physical layer the `CC2420` radio transceiver is used, which comes with the TmoteSky sensor nodes. This radio transceiver handles the whole radio communication, and is accessed from the `msp430` MCU via a SPI[1] interface. On the data link layer two different MAC protocols are used: `NullMAC` and `ContikiMAC`. For sharing a transmission medium between different clients, the Carrier Sense Multiple Access (CSMA) method with collision avoidance is used. For more information on the used MAC protocols, see chapter 2.3.4.

On the network layer the µIP stack is used, which comes with the Contiki OS. The µIP

[1]The Serial Peripheral Interface (SPI) is a synchronous serial bus system, which operates by the master/slave principle.

stack runs on top of RIME, which is a lightweight layered communication stack for WSNs [13]. Chapter 2.3.2 provides more details on µIP and RIME. The main task of the network layer is routing different packets between sensor nodes that have no direct connection.

On the transport layer UDP is used. UDP is a very simple, stateless and unreliable transport protocol, it does not guarantee the arrival of a packet at the designated destination. There exists no real end-to-end connection between two communication partners. If a packet is lost, the application itself has to take care about the retransmission. The simplicity of UDP has the advantage of a very small overhead and is therefore popular in real-time and embedded systems. UDP simply adds a small header to an IP packet, which consists of four attributes: a source- and destination-port, the length of the payload (including the header) and a checksum over the whole packet.

The implemented communication protocols directly use UDP to transmit data. They take care of the retransmission of lost data. Additionally they also handle the fragmentation and reassembly of payloads bigger than the maximum transmission unit (MTU).

3.2 NACK-Based Reliability Mechanism

Since UDP is unreliable, but offers a very small overhead, we had to implement our own reliability mechanism to make sure that lost packets are retransmitted. Implementing an own reliability mechanism brings the advantage of better control of the retransmission handling, and therefore allows optimization of the whole process to fit our needs.

The standard reliability mechanism of most common transport protocols uses short messages, so-called "acknowledgments" (ACKs), to confirm the reception of a packet. Every received packet generates a corresponding ACK. Obviously this method generates a lot of control traffic, which is responsible for significant overhead.

One of our main goals was, to implement the chosen communication protocols in such a way, that they use as less energy as possible, since energy is the most constrained resource of sensor nodes. To reach that goal, another approach generating less control traffic had to be found to take care of lost packets:

First the receiver has to be informed about the amount of packets to be transmitted. Additionally each packet is assigned a unique identifier (UID). If the receiver receives a packet, it makes a note that the packet with UID x has been received. After reception of the last packet, the receiver checks for lost packets, and if necessary, requests them by sending a "negative acknowledge" (NACK) message. If the receiver received all packets, it finally sends an ACK message. The usage of NACKs significantly reduces the amount of control traffic, since not every successful packet reception has to be acknowledged, but instead only the missing packets have to be requested.

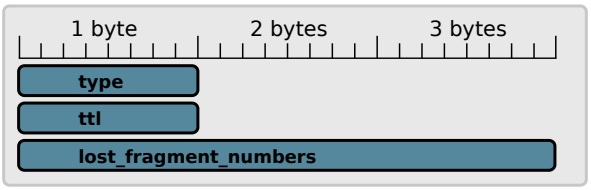

Figure 3.2: NACK message

Figure 3.2 shows the structure of a NACK message. The `type` attribute is a integer of one byte width, which is used to distinguish different messages types. The second attribute, `ttl` is an abbreviation for "Time to Live" (TTL) and is also an integer of one byte width. The TTL attribute is used as a loop prevention mechanism. Every time a NACK message is forwarded, the TTL value is decremented. If the TTL value reaches zero, the NACK message is dropped. The last attribute, `lost_fragment_numbers`, consists of three one byte integers, indicating the lost packets. Accordingly, up to three lost packets can be requested with one single NACK message. Referencing chapter 1.2, no more than 15 unique packets are transmitted. We therefore made the design decision to request three packets with one NACK message.

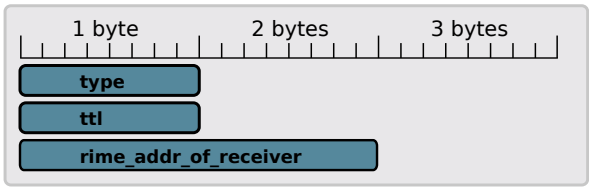

Figure 3.3: ACK message

Figure 3.3 shows the structure of an ACK message, which is quite similar to a NACK message. The first two attributes are the same, only the third attribute differs. The attribute `rime_addr_of_receiver` consists of two one byte integers, containing the RIME address of the receiver sending the ACK message. This message indicates the successful reception of all packets.

3.3 Caching Strategies

The caching strategy of a communication protocol can be crucial for the overall performance: The time for transmitting data is proportional to the energy used. To minimize the energy usage, we have to minimize the transmission time. Obviously minimizing the transmission time is only one, but important factor among others influencing the energy consumption. As described in chapter 1.2, three different caching strategies are used on the intermediate nodes:

- no caching
- caching of every fragment
- proactive caching

The caching of the different fragments takes place on the intermediate nodes, which are found on the path from the sender to the receivers.

The first caching strategy is very simple: The intermediate nodes do not cache any fragments. This implies that every NACK message has to be replied by the sender itself. Using the second

caching strategy, every intermediate node caches every unique fragment. An intermediate node does not cache a fragment twice, it caches the fragment only if it is not already cached before.

The size of the cache is an important design decision, especially in memory-constrained environments like WSNs. Since the maximum size of the payload is equal to 1000 bytes, the maximum size of the cache would be 1000 bytes, if all unique packets would be cached. The `msp430` MCU, which is used by the TmoteSky sensor nodes, provides 10KB of main memory. Since the Contiki OS needs about 2KB of main memory, there is approximately 8KB free for usage by the user. We therefore made the design decision to make the cache 1000 bytes big to optimize the transmission time as much as possible.

The third caching strategy is called proactive caching, because the intermediate nodes are allowed to request lost fragments autonomously. A intermediate node only requests fragments, if it did not receive a fragment for a certain time.

3.4 Protocols

This section presents the chosen designs for implementing Flooding, MPR and Directed Diffusion. It's important to mention that the chosen designs used to implement these protocols differ from the original design. First, we used UDP as transport protocol, whereas mostly TCP is used. The usage of UDP instead of TCP as transport protocol implied the implementation of an own reliability mechanism, which changes the behavior of a protocol massively. Secondly, different caching strategies are used. The original design mostly uses only one specific caching strategy. Thirdly, we adapted the designs of the protocols to our needs: In our scenario, the data shall be distributed to more than one receiver. Therefore we had to adapt especially the design of Directed Diffusion to a sender-driven approach.

3.4.1 Communication Phases

The three implemented communication protocols can all be split into four communication phases, which are illustrated by Figure 3.4. A special phase called *initialization phase* is only used by MPR and Directed Diffusion, because these protocols need to prepare the WSN for the data transmission.

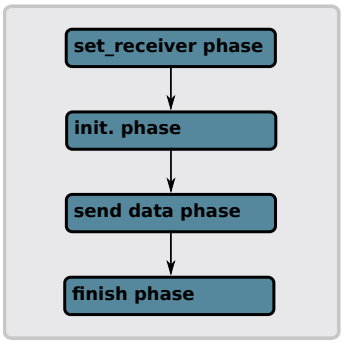

Figure 3.4: The four communication phases

The first phase of the data transmission is the *set_receiver phase*. That is the beginning of the communication, where the sender informs the receiver that it will receive data. This is a prerequisite, because if a sensor node does not know that it will receive data, it can not request the data if all fragments are lost on their way to the receiver. If the payload consists of only one packet, it is very likely that this will happen. Figure 3.5 illustrates the structure of the `set_receiver` message. The third attribute `receivers` consists of three integers of one byte width each. This attribute stores the last part of the RIME addresses of the three receivers.

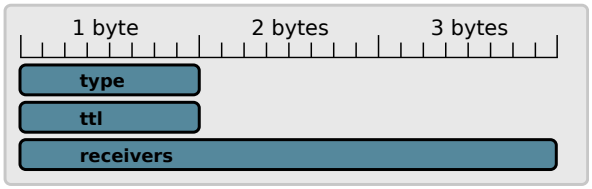

Figure 3.5: The set receiver message

The second phase is the *initialization phase*, which is used only by MPR and Directed Diffusion. In this phase the protocols prepare to send data: MPR calculates the Multipoint Relay set, whereas Directed Diffusion sets up the routing gradients.

The third phase, the *send data phase*, is the actual data transmission phase.
Finally, in the *finish phase* the sender informs all participating sensor nodes of the successful data transmission. This phase is necessary to get accurate evaluation data: the energest values and the amounts of collisions and sent frames. Without this phase all intermediate sensor nodes would have to send this evaluation relevant data periodically to their attached mesh node, which would bias the experiment data significantly: Since the sensor nodes are attached to a mesh node via a RS232 connection, the data transmission to the mesh node is quite slow. While transmitting this data, the sensor node can not react to incoming packets. Additionally, if the buffer for incoming packets is full, the packets are dropped. Therefore, every unnecessary usage of the RS232 connection slows down data transmission and increases energy consumption of the sensor node. To inform all participating sensor nodes of the successful data transmission, the sender broadcasts an ACK message with the attribute `rime_addr_of_receiver` set to zero.

3.4.2 Flooding

Flooding is a very basic communication protocol. The sender starts to broadcast the first packet and each intermediate node re-broadcasts the packet until it reaches the receiver. Obviously some sort of loop prevention is needed, otherwise packets are forwarded infinitely, resulting in a broadcast storm. Figure 3.6 illustrates this situation.

The loop prevention mechanism for data messages chosen for this implementation consists of a simple, but effective measure: *Each data message is forwarded only once by each intermediate node.*

If each node caches all data messages, intermediate nodes can reply all NACK messages sent to the sender node. This measure works perfectly when all the data is cached, but it has it is weakness when no caching is used: When a receiver is requesting a data message, it sends a

NACK message. This NACK message is forwarded until it reaches the sender node. The sender node then resends the requested data messages, but no intermediate node will forward the resent data messages, as they already forwarded them.

For all other types of messages (like ACKs and NACKs) a TTL mechanism is used to prevent loops, since only data messages are cached and as mentioned above, without any caching our first approach would not work.

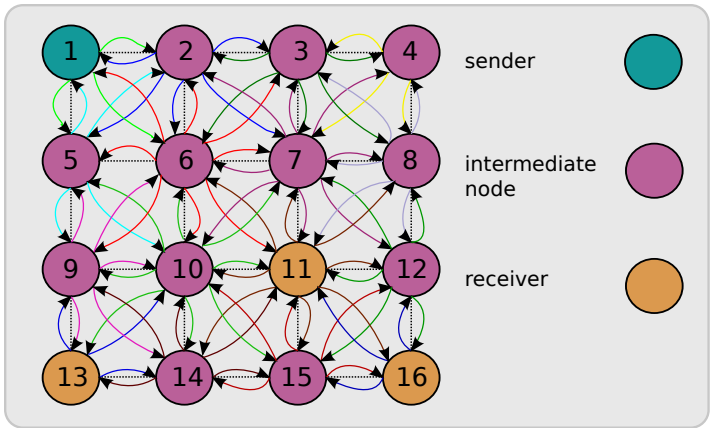

Figure 3.6: Broadcast storm

Through the heavy usage of broadcast communication, there are a lot of sensor nodes which want to send data at the same time. This results in a rival behavior of the different sensor nodes. If two sensor nodes try to send data at the same time, a collision occurs. Without the usage of a resend mechanism like CSMA, the packets are dropped. The CSMA implementation of Contiki tries to resend a packet six times. To minimize the occurrence of collisions, a random backoff time is used, which each sensor node waits before sending any data. This random backoff time is static and is configured individually for every kind of message.

Figure 3.6 shows the situation, which occurs when Flooding is used to transmit the data. There is a lot of ongoing communication, resulting in a lot of collisions and a slow data transmission. As mentioned in chapter 1.2, splitting the payload into smaller fragments is necessary, therefore the term “fragments” is used synonymously for the term “packets”.

To transmit individual fragments, the data message is used. Figure 3.7 illustrates the used data message.

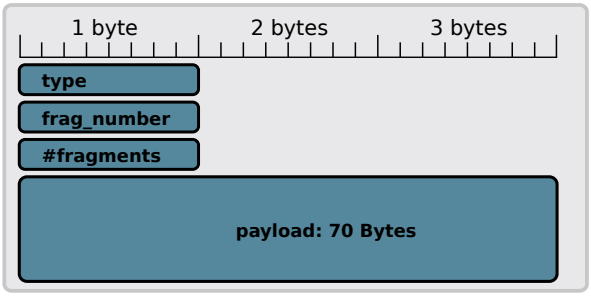

Figure 3.7: The data message

The first attribute is the same as in the already discussed messages types. The attribute `frag_number` is the UID, used to distinguish different fragments. The UID starts at one and is incremented for every unique fragment. This unique identifier is important for the caching mechanism and for loop prevention. The third attribute, `#fragments` denotes the number of fragments and is equal to the highest UID. The number of fragments is used by the receivers to check whether the last fragment was received. If so, the receivers will check for lost fragments and request them. The last attribute, `payload`, is used to transport the data.

Figure 3.8 illustrates the sequence diagram for the *send data phase* of Flooding with no caching.

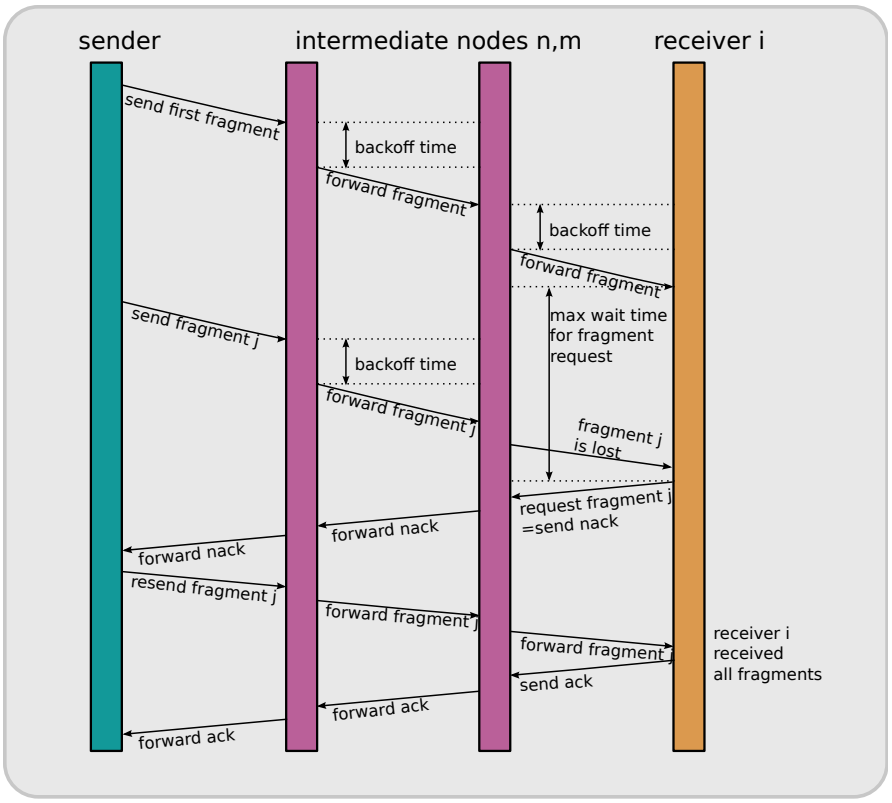

Figure 3.8: The sequence diagram for Flooding with no caching

The sender starts data transmission by broadcasting the first fragment. Every node, which receives this fragment waits for a random backoff time before re-broadcasting it. After waiting for a fixed time, the sender continues broadcasting the next fragment, until all fragments were sent. If a receiver does not receive a fragment for a certain interval, it checks for missing fragments and requests them by sending a NACK message. Since no fragments are cached, the NACK message has to travel back to the sender. The sender then re-broadcasts the requested fragments. The receivers check periodically whether they received all fragments. If so, they

signal the success of the data transmission by sending an ACK message to the sender. After receiving a ACK message from all receivers, the sender knows the data transmission succeeded.

Without any caching of fragments, the retransmission of lost fragments takes a long time, since the sent NACK message has to travel back to the sender. If the intermediate nodes cache fragments, it is possible for each node on the route of the NACK message to retransmit the fragments, assuming the requested fragments are cached. With a proactive caching strategy, the intermediate nodes behave similarly as the receivers: If an intermediate node did not receive a fragment for a certain interval, it starts requesting the missing fragments until it receives the desired data.

3.4.3 Multipoint Relay

Multipoint Relay (MPR) is a communication protocol, which is also broadcast-based like Flooding. In contrast to Flooding, MPR tries to reduce the number of forwarding intermediate nodes with the goal of reducing the amount of collisions. The lesser the amount of collisions, the faster is the data transmission. Every node calculates its own MPR set, that is the set of its one-hop neighbors, reaching most of its two-hop neighbors. To reach all of the two-hop neighbors, the MPR set has to be increased by the remaining one-hop neighbors reaching the missing two-hop neighbors.

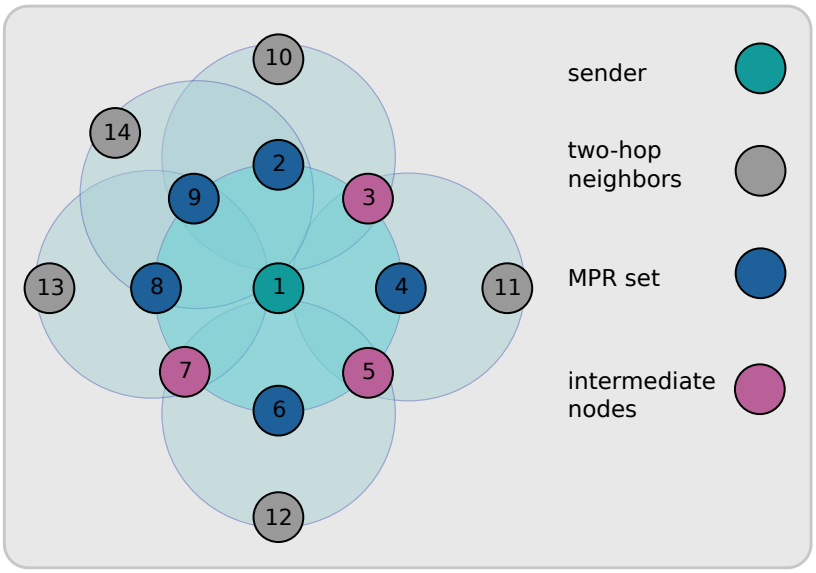

Figure 3.9: MPR set

Figure 3.9 shows an exemplary WSN with 14 sensor nodes. The MPR set for node 1 is denoted by the blue nodes. Through this set of nodes, it is possible for node 1 to reach almost all of its two-hop neighbors. Since node 14 is not reachable from this MPR set, the set has to be increased by node 9. The purple nodes are intermediate nodes, which do not forward any data. These nodes are not involved in any radio communication, representing the main difference between MPR and Flooding: Using MPR as communication protocol, not all intermediate nodes

forward data, but only a small subset of them. The calculation of the MPR sets is done at the end of the *initialization phase*. Before these sets can be calculated, every node has to determine his one- and two-hop neighborhood. For this purpose, every node broadcasts hello messages for a certain interval.

Figure 3.10 illustrates the structure of a hello message.

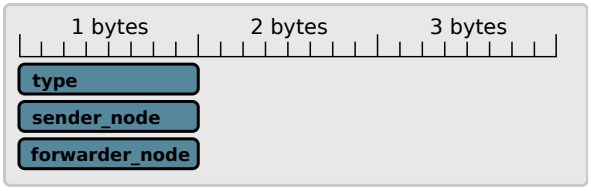

Figure 3.10: The hello message

The second attribute, `sender_node` contains the last part of the RIME address of this node, which re-broadcasted the hello message for the *first* time. The third attribute, `forwarder_node` is similar as the second attribute, with the difference, that it contains the last part the RIME address of this node, which re-broadcasted the hello message for the *second* time.

Figure 3.11 visualizes the neighborhood discovery using hello messages.

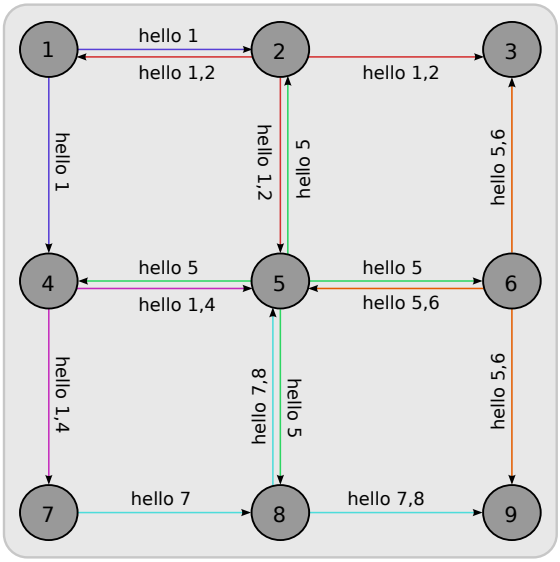

Figure 3.11: Neighborhood discovery using hello messages

If a node receives a hello message, it first checks if the `forwarder_node` attribute is set and if the `sender_node` value is not equal to the last part of its RIME address. If so, it knows that it received a hello message from a two-hop neighbor, since this hello message was forwarded only twice. If the `forwarder_node` attribute is not set, the node knows that it received a hello message from a one-hop neighbor. It stores the last part of its RIME address

to the `forwarder_node` attribute of the hello message and re-broadcasts the message. Every node collects each hello message received, and and stores them accordingly to their properties: If only the `sender_node` attribute is set, the message is added to the set of one-hop neighbors. If the `sender_node` and the `forwarder_node` attribute is set and the `sender_node` attribute is not equal to the last part of the RIME address of the node, the message is added to the set of two-hop neighbors.

Using the obtained neighborhood information, every node can calculate its own MPR set using the heuristic proposed by [6].

Figure 3.12 shows the pseudo code of this heuristic, where *MPR(x)* denotes the MPR set, *N(x)* the one-hop neighborhood and $N^2(x)$ the two-hop neighborhood of node *x*. Once the MPR sets are determined, every node broadcasts a forwarder message to inform the chosen nodes that they are allowed to forward data.

1. Start with an empty multipoint relay set *MPR(x)*
2. First select those one-hop neighbor nodes in *N(x)* as multipoint relays which are the only neighbor of some node in $N^2(x)$, and add these one-hop neighbor nodes to the multipoint relay set *MPR(x)*
3. While there still exist some node in $N^2(x)$ which is not covered by the multipoint relay set *MPR(x)*:
   (a) For each node in *N(x)* which is not in *MPR(x)*, compute the number of nodes that it covers among the uncovered nodes in the set $N^2(x)$
   (b) Add that node of *N(x)* in *MPR(x)* for which this number is maximum.

Figure 3.12: Heuristic used to evaluate the MPR set

Figure 3.13 illustrates the structure of a forwarder message. The attribute `mprs` consists of ten one byte integers, containing the last part of the RIME address of these nodes, which are members of the MPR set.

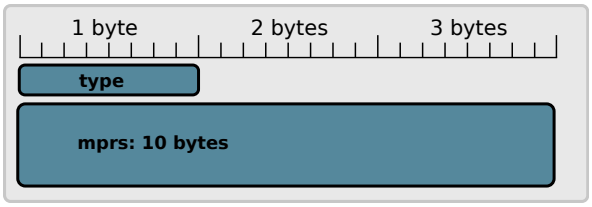

Figure 3.13: The forwarder message

At the end of the *initialization phase*, the WSN is prepared for data transmission: All sensor nodes, that are member of a MPR set are forwarding data. The other intermediate nodes are not

involved in any radio communication. The *send data phase* and the *finish phase* are equal with the corresponding phases of Flooding.

3.4.4 Directed Diffusion

Directed Diffusion (DD) [5] is a communication protocol, which is quite different to the previously discussed protocols. Its main difference is that the data is *not broadcasted*, but instead transmitted by UDP unicast hop-by-hop. For initializing data transmission, the receivers broadcast interest messages towards the sender. That is done during the *initialization phase*.

Figure 3.14 illustrates the structure of an interest message.

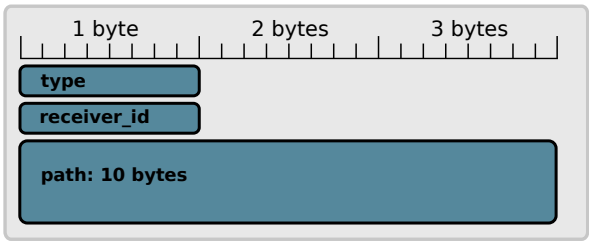

Figure 3.14: The interest message

The attribute `receiver_id` denotes the last part of the RIME address of the receiver, which initially sent the interest. The third attribute `path` contains the last part of the RIME address of every sensor node on path of the interest from the receiver to the sender. Since the interest messages are broadcasted, a loop prevention mechanism is needed again: If an intermediate node receives an interest message, it checks the path of the message for its own RIME address. In the case of a match the node knows that it already forwarded this interest message, and can therefore drop it. The sender collects all received interest messages and after a certain time it starts processing them:

1. the path stored in the interest messages is reversed
2. the hops of every path are counted
3. depending on a evaluation metric, the sender chooses one particular path for every receiver

The number of hops was chosen as evaluation metric: The shorter the path, the less unicast transmissions are needed for the fragment to reach the receiver. There are also other possible evaluation metrics, as example the occurrence of the different paths could be counted, assuming that a path that occurs more often as others is more reliable. The path that occurs the most would be chosen. After choosing one particular path to every receiver, the sender begins to *reinforce* these paths.

Figure 3.15 shows the sequence diagram for the *initialization phase* of Directed Diffusion.

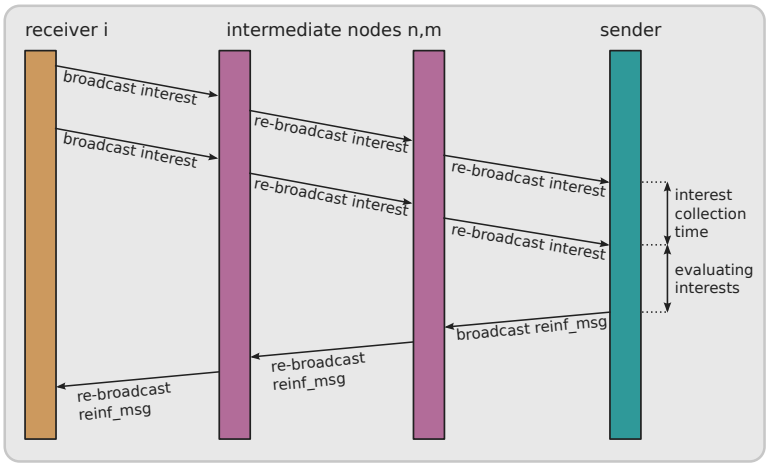

Figure 3.15: The sequence diagram for the *initialization phase* of Directed Diffusion

The goal of the reinforcement of a path is to setup the gradients on every intermediate node found on the path to the receiver. In our context a gradient is a data structure containing routing information. A intermediate node can have multiple different gradients: one for every particular receiver. To reinforce the chosen paths, the sender broadcasts the reinforcement message.

Figure 3.16 illustrates the structure of the reinforcement message.

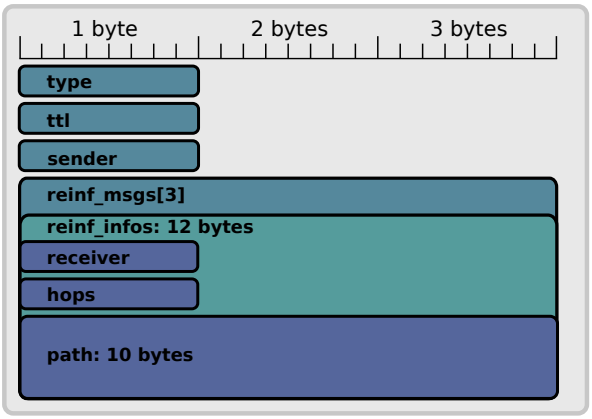

Figure 3.16: The reinforcement message

The second attribute `ttl` is a Time to Live value, which is decreased every time the message is forwarded. That is necessary, because the sender *is broadcasting* the reinforcement message. Broadcasting the reinforcement message is a design decision we made. The alternative would be to send the reinforcement message via UDP unicast along the chosen path to the receiver. We did not choose this approach because UDP is not reliable: The setup of the gradients is the single point of failure of Directed Diffusion. If a packet containing the reinforcement message is lost, the gradients can not be established. Without adequate routing information it is impossible for the nodes on the chosen path to forward the fragments to the next hop, and the whole data transmission would fail. Therefore, it is more reliable to broadcast the reinforcement message

instead to send it via UDP unicast. To optimize broadcasting of the reinforcement message, all necessary information for the reinforcement of all paths was embedded into one distinct reinforcement message. The fourth attribute `reinf_msgs` contains the reinforcement informations for all receivers and consists in our case of three attributes (since we have only three receivers), with the name `reinf_infos`. The attribute `reinf_infos` consists again of three attributes: `receiver`, `hops` and `path`. The attribute `receiver` contains the last part of the RIME address of the receiver, `hops` contains the number of hops for the chosen path and `path` contains the path to the receiver.

If an intermediate nodes receives a reinforcement message, it first checks if it already processed a reinforcement message. If so, it does not process the message again, since it has to process this message only once, because all necessary information to setup the gradients was embedded into the reinforcement message. Next the node checks for every receiver if it finds the last part of *its* RIME address in the `path` attribute. If so, the node knows that it has to establish a gradient between its predecessor and its successor found in the `path` attribute. This gradient means the following: For one specific receiver all data towards this receiver is forwarded to the successor and all data towards the sender is forwarded to the predecessor. After the processing of this message the node checks the `ttl` value. If it is greater than zero it rebroadcasts the message, otherwise not.

When all intermediate nodes on the path to every receiver established their gradients, the *initialization phase* is over. There exists now one specific path to every receiver, which is characterized by the gradients found on its intermediate nodes.

Figure 3.17 illustrates this situation.

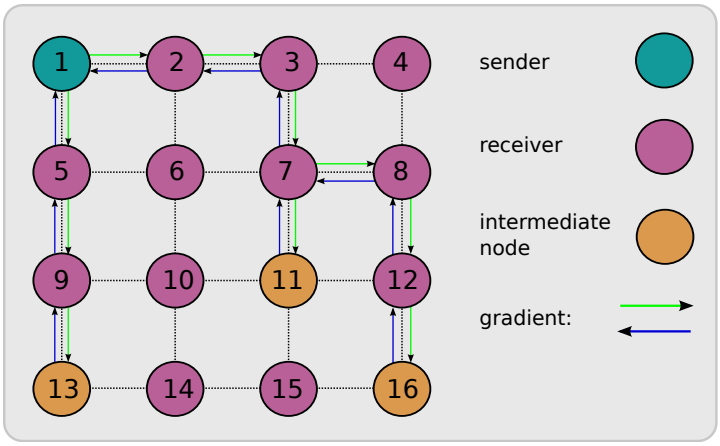

Figure 3.17: Gradient setup

A pair of a blue and a green arrow symbolize a gradient, consisting of routing information for the *next hop forward* and the *next hop backward*. An intermediate node can have more than one gradient, if this node lies on more than one path. Node 7 is an example for this situation. Along the blue path an interest was broadcasted and it is used to send NACK and ACK messages. The green path is used to transmit fragments, since fragments are transmitted only in the direction from the sender to the receiver. After finishing of the *initialization phase*, the sender begins

with the data transmission. The *send data phase* is almost equal with the corresponding phase of Flooding and MPR, with the little but significant difference that fragments are transmitted by UDP unicast and *not* by UDP broadcast to the next hop.

In summary, Directed Diffusion can be described as a communication protocol, which first discovers its neighborhood using interest messages. By the diffusion of these interest messages through the WSN towards the sender, Directed Diffusion collects indirectly routing information in the form of paths from the receiver to the sender. The sender then processes this routing information and sets up the gradients towards the receivers. The actual data transmission is a usual UDP unicast transmission, with the difference that Directed Diffusion does not rely on static routing tables, since Directed Diffusion uses the gradients to forward the fragments to the next hop. Through the independence from static and preconfigured routing tables, Directed Diffusion can transmit data via UDP unicast without prior knowledge of the WSN topology.

An other significant difference between a usual UDP unicast transmission and the send data phase of Directed Diffusion is following: In our implementation, Directed Diffusion is an application layer protocol (since it uses UDP as transport protocol) , meaning that each fragment has to travel up the whole protocol stack to the application layer for every hop. Compared with a usual UDP unicast transmission, where a packet on the way to its receiver does not travel higher as the network layer, this overhead is not negligible.

Chapter 4

Implementation

Flooding, MPR and Directed Diffusion were implemented on top of Contiki OS. During the implementation of these communication protocols, we paid attention to develop platform independent code. This was accomplished by using only high-level application programming interfaces (APIs) provided by the Contiki OS and the avoidance of the usage of `TmoteSky` specific features. To develop the code, a Vmware image of a prepared Ubuntu Linux was used, which was provided by the developers of Contiki. This image contains all necessary *preconfigured* software tools for developing and compiling code for the TmoteSky platform. The functionality of Flooding, MPR and Directed Diffusion was split into two parts: a sender part and a receiver part. Therefore, two different software images were used for flashing the TmoteSky sensor nodes: the sender image, which initiates the data transmission, and the receiver image, which is responsible for receiving and forwarding the data. To illustrate the structure of the developed code, we are using pseudo code listings. The goal of these listings is not to exactly map the developed code, but instead to give an abstract view of the main ideas behind the code. Therefore, some constrains and boundary checking code was omitted in the pseudo code listings.

4.1 Caching

To implement the cache discussed in chapter 3.3, a simple buffer of the payload size was used to model the cache, since all fragments are cached. To cache a fragment, an intermediate node simply copies the data of the fragment to the right position in the cache buffer. This position in the cache buffer is characterized by a start byte, which can be derived from the UID of the fragment. The calculation and the copying of the data to the cache buffer is implemented as follows:

Listing 4.1: Caching of a fragment

```
1 end_byte    = (UID       * sizeof(fagment.data) ) -1;
2 start_byte  = (end_byte - sizeof(fagment.data) ) +1;
3 memcpy(&(cache_buf[start_byte]), &(fragment.data), sizeof(fagment.data));
4 received_fragments[fragment.fragment_number] = 'R';
```

Additionally, the buffer `received_fragments` was used to keep track of the received fragments. To check if a certain fragment with UID x was cached, the value of `received_fragments[x]` has to be compared with the char `R`. If these two values are equal, then the fragment x was cached. For accessing the data of a cached fragment with UID x, the value of the `start_byte` of fragment x has to be calculated. Using this `start_byte`, the data can be copied with the right offset to a arbitrary buffer.

4.2 Payload Splitting

The splitting of the payload of 1000 bytes to smaller fragments of 70 bytes each is accomplished by the function `buildFragment(uint8_t UID)`. This function builds a `data_message` (fragment) with the UID supplied as function argument. To get the right data, this function uses the `start_byte` derived from the UID to copy 70 bytes from the payload buffer to the payload buffer of the `data_message`.

4.3 Packet Queue

As discussed in chapter 2.3.2, the µIP TCP/IP stack of Contiki uses a single buffer for incoming and outgoing packets. Therefore µIP doesn't have a queue for incoming packets and only stores one packet. That's problematic because of the preconfigured backoff time a node waits before it actually transmits a packet. During this time period, the node isn't able to receive new packets, since the processing of the current package isn't finished and µIP doesn't overwrite the global packet buffer. Depending on the used MAC protocol, which also may or not cache frames, these new packets are dropped.

We, therefore, developed a packet queue, which caches a preconfigured number of packets. This packet queue was implemented as a separate Contiki process, which acts as a proxy between the communication protocol and µIP. The communication protocol does not talk directly to µIP, instead it uses the packet queue for receiving new packets. The interaction between the communication protocol and the packet queue can be characterized as follows:

First, the process of the communication protocol has to tell the packet queue that it wants to use the queue. The process does that by executing the `attachToQueue(handle h)` function. This function stores the process ID of the process of the communication protocol in the data structure `h` of the type `handle`. Next, the `attachToQueue` function starts the packet queue process, which is listening for `tcp_ip` events posted by µIP. If new packets arrive, µIP posts the `tcp_ip` event, signaling that a new packet was copied into the global packet buffer. The packet queue now checks for a free packet slot. If so, the new packet in the global packet buffer is copied to this slot, if not, the packet is dropped. The packet queue now posts the `packet_queue` event to the process with the ID stored in the data structure `handle`. The process of the communication protocol has to listen for the event `packet_event`. If such an event arrives, the process uses the `getPacket` function to access the new packet. This function returns a pointer to the data of the new packet.

Besides the buffering packets, the packet queue is used for aggregation of data messages.

That is especially useful for broadcast-based communication protocols, since with this approach each node receives the same packet many times. Using a packet queue results in the situation of a queue filled with a lot of duplicated packets. Data aggregation is used to eliminate duplicated packets from the packet queue. That is accomplished by iterating over the queue and comparing each packet. If two packets are equal, one of them is dropped.

4.4 Protocols

4.4.1 Flooding

As mentioned above, Flooding consists of two parts: A sender part and a receiver part. Each part is running as an independent Contiki process on its sensor node. The sender process can be described in pseudo code as follows:

Listing 4.2: Sender process of Flooding

```
1  PROCESS_BEGIN()
2    attach this process to the packet queue
3    send the set_receiver message
4    PROCESS_WAIT for 5 seconds
5
6    for each fragment i do
7      PROCESS_WAIT for a random backoff time
8      buildFragment(i)
9      broadcast_fragment(i)
10
11   while(1){
12     PROCESS_WAIT for packet_event or periodic timer event
13     get the new packet from the packet queue
14
15     if received new data:
16       switch(message_type)
17          case nack_message: process nack -> resend requested fragment
18          case ack_message:  process ack  -> note that receiver finished
19
20     else
21       if all receivers finished:
22         send finished message
23   }
24 PROCESS_END()
```

Listing 4.2 shows the three communication phases, which this process passes:

The first phase is the *set_receiver phase*, which is used by the sender to inform the chosen receivers for incoming data. The next phase is the *send data phase*, which is used to send all fragments at once. While sending these fragments, the sender doesn't respond to incoming messages.

After the transmission of all fragments, the sender begins to respond to incoming messages. Depending on the type of the message, the sender will resend the requested fragments or make note that a certain receiver received all fragments. If the sender received

an ACK message from all receivers, the *finish phase* starts. The sender knows now that all receivers received all fragments and starts broadcasting an ACK message with the attribute `rime_addr_of_receiver` set to zero. In line 12 of the listing it can be seen that the process blocks and waits for a `packet_event` or a `periodic_timer_event`. Listening for the `periodic_timer_event` is necessary for waking up the sender process periodically, since without this event this process is waked up only when a new packet arrives. But the sender has periodically to check whether it received all ACKs.

Listing 4.3: Receiver process of Flooding

```
1  PROCESS_BEGIN()
2  attach this process to the packet queue
3
4  while(1){
5    PROCESS_WAIT for packet_event or periodic timer event
6    get the new packet from the packet queue
7
8    if appstate != finished
9      if received new data:
10       switch(message_type)
11          case set_receiver_message:
12             if i'am a receiver:
13                isReceiver=1
14             if ttl > 0: decrement ttl and forward
15          case data_message:
16             if not already forwarded:
17                if caching is on: add payload to cache
18                if isReceiver == 0: forward
19          case nack_message:
20             if requested fragment is in the cache
21               resend fragemnt
22             else
23               if ttl > 0: decrement ttl and forward
24          case ack_message:
25               if rime_addr_of_receiver == 0: print statistics
26               if ttl > 0: decremnt ttl and forward
27     else
28       if all fragments received and isReceiver == 1
29         appstate=finished
30       else
31         if i'am a receiver and not received a packet for a certain time:
32           request missing fragments
33   else
34       send ack
35
36 }
37 PROCESS_END()
```

Listing 4.3 illustrates the pseudo code for the receiver part of Flooding. It can be seen that this process also consists of an infinite while loop and is waked up after the reception of a `packet_queue_event` or a `periodic_timer_event` (line 5). If a new packet arrives,

it is treated depending on its type by the switch statement (line 10). If no packets arrive for a certain time, the first three fragments (that are the fragments with UID 1,2 and 3) are requested (line 32). The receiver process maintains a state variable called `appstate`, which indicates whether all fragments were received. If the value of `appstate` is equal to "finished", an ACK message is broadcasted. Line 16 illustrates the used loop prevention mechanism: A fragment is forwarded only for the first time. Afterwards it is resent from the cache (line 20).

4.4.2 Multipoint Relay

As discussed in chapter 3.4.3, MPR works similarly as Flooding. The difference is that only *some* intermediate nodes are allowed to rebroadcast the received packets.

Listing 4.4: Sender process of MPR

```
1  PROCESS_BEGIN()
2    attach this process to the packet queue
3
4    while(1){
5      PROCESS_WAIT for packet_event or periodic timer event
6      get the new packet from the packet queue
7
8      if received new data:
9        switch(message_type)
10          case hello_message:
11             if received from one-hop neighbour:
12                add to one-hop neigbour set
13                set forwarder_node to my node_id
14                forward hello_message
15             if received from two-hop neighbour:
16                add to two-hop neighbour set
17          case nack_message:
18             resend requested fragment
19          case ack_message:
20             note that receiver finished
21     else
22       if all receivers finished:
23         send finished_message
24       if "hello message collect time" is over and mpr sets are not already
              evaluated:
25          evaluate mprs
26          send forwarder_message
27          send the set_receiver message
28          PROCESS_WAIT for 5 seconds
29
30          for each fragment i do
31            PROCESS_WAIT for a random backoff time
32            buildFragment(i)
33            broadcast_fragment(i)
34   }
35 PROCESS_END()
```

Listing 4.4 illustrates the pseudo code for the sender part of MPR. It is noticeable that the `set_receiver_message` and the actual data are not sent before the "hello message collect time" is over and the MPR set was determined (line 24). During this "collect time" all `hello_messages` are collected or forwarded (line 10-16). If this time expired, the MPR set for the sender node will be evaluated. For the sake of simplicity, the evaluation of the MPR set is described in the listing 4.5.

After broadcasting the `forwarder_message` and the `set_receiver_message`, the sender will wait for 5 seconds until it starts to broadcast one fragment after an other (line 30-33). The sender now waits for incoming NACK or ACK messages. If a NACK messages arrives, the requested fragment will be resent (line 18). If an ACK messages arrives, the sender will make a note that the mentioned receiver received all fragments (line 20).

The `periodic_timer_event` wakes up the sender process periodically. If meanwhile no new data arrived, the sender will check if all receivers finished (line 22-23).

Listing 4.5: MPR set evaluation

```
1  list evaluate_mpr(list one_hop_neighbours, list two_hop_neighbours){
2
3  for each node i do:
4    for all hello_messages h_2 in two_hop_neighbours do:
5
6      if the sender_node attribute of h_2 is equal to the node i:
7
8        for all hello_messages h_1 in one_hop_neighbours do:
9           if the sender_node attribute of h_1 is equal to the forwarder
                attribute of h_2:
10             forwarderCount[h_2.forwarder_node]++
11
12   find the element with the highest value of forwarderCount.
13   mprCandidats[i] is equal to the index of this element.
14
15   remove all elements from forwarderCount
16
17
18 for each node i do:
19   remove all dumplicated elements from mprCandidats.
20
21 return list(mprCanditats)
22 }
```

Listing 4.5 shows the pseudo code for the heuristic, which is used for evaluating the MPR set. This method first picks out all `hello_messages` sent by node i, which were forwarded *only once* (line 6). Then it checks if these `hello_messages` were forwarded by a one-hop neighbor (line 8-9). If so, this one-hop neighbor is a possible candidate for the MPR set.

Therefore, the method makes a note of how many `hello_messages` were forwarded by this particular node. That is accomplished by incrementing *the value at the index of the node id* of the array `forwarderCount` (line 10). The index of the array `forwarderCount` denotes the last part of the RIME address of the node, which forwarded the `hello_message`. The value at this index denotes *how many* `hello_messages` were forwarded by this node. That one-hop

array index	*array value*
node id	#forwarded msgs
1	0
2	0
3	5
4	15
5	4
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0

Table 4.1: Exemplary content of the forwarderCount array

neighbor is chosen as MPR node for node i, which forwarded *the most* `hello_messages`.

Therefore, the method compares all values of the array `forwarderCount`, and searches the highest value (line 12). The index of this value is the last part of the RIME address of the MPR node for node i. This result is stored to the array `mprCadidats` (line 13). Now the next node i is processed and therefore all elements of the array `forwarderCount` are deleted (line 15). After removing all duplicated elements from the array `mprCandidats`, the MPR set was evaluated (line 18-19).

Table 4.1 shows an exemplary content of the `forwarderCount` array of node 1, for node 11 of the topology illustrated by figure 3.9. It can be seen that node 1 received `hello_messages` forwarded by node 3,4 and 5. Node 4 forwarded the most `hello_messages`, therefore, node 4 is added to the MPR set.

Listing 4.6: Receiver process of MPR

```
1  PROCESS_BEGIN()
2  attach this process to the packet queue
3  while(1){
4
5    send hello_message
6    PROCESS_WAIT for packet_event
7
8    if received new data:
9        if message_type = hello_message
10          if received from one-hop neighbour:
11             add to one-hop neigbour set
12             set forwarder_node to my node_id
```

```
13              forward hello_message
14           if received from two-hop neighbour:
15              add to two-hop neighbour set
16      if "hello message collect time is over" and mpr sets are not already
            evaluated:
17        evaluate mprs
18        send forwarder_message
19        send the set_receiver message
20        break;
21  }
22
23  while(1){
24    PROCESS_WAIT for packet_event or periodic timer event
25    get the new packet from the packet queue
26
27    if appstate != finished
28      if received new data:
29        switch(message_type)
30           case set_receiver_message:
31              if i'am a receiver:
32                 isReceiver=1
33              if ttl > 0: decrement ttl and forward if i'am a forwarder node
34           case data_message:
35              if not already forwarded:
36                 if caching is on: add payload to cache
37                 if isReceiver == 0: forward if i'am a forwarder node
38           case nack_message:
39              if requested fragment is in the cache
40                resend fragemnt
41              else
42                if ttl > 0: decrement ttl and forward if i'am a forwarder
                      node
43           case ack_message:
44                if rime_addr_of_receiver == 0: print statistics
45                if ttl > 0: decremnt ttl and forward if i'am a forwarder
                      node
46      else
47        if all fragments received and isReceiver == 1
48          appstate=finished
49        else
50          if i'am a receiver and not received a packet for a certain time:
51            request missing fragments
52    else
53        send ack
54  }
55  PROCESS_END()
```

Listing 4.6 illustrates the pseudo code for the receiver part of MPR. At the first glance it can be seen that this process consists of two *while(1)* loops. The first *while(1)* loop is used for the *initialization phase* of MPR (line 3-19). During this phase, the intermediate nodes send and collect `hello_messages`. A node starts by sending a `hello_message` and waits until it receives one (line 5-6). If it receives a `hello_message`, it checks if it received the message

from a one-hop-neighbor or from a two-hop-neighbor. This check is possible, because every time when a `hello_message` is forwarded, the `forwarder_node` attribute is set by the node that forwarded the message. If the message is from a one-hop neighbor, then the `forwarder_node` attribute is set and the message is forwarded. If the message is from a two-hop neighbor, then the message is only added to the set of two-hop neighbors (line 9-15). If the "hello message collect time" is over, the MPR set will be evaluated (line 16-17).

First, an intermediate node broadcasts the `forwarder_message` and then `set_receiver_message`. Afterwards it quits the *while(1)* loop using the *break* statement (line 18-20). Now, the second *while(1)* loop is used. This part of the process of quite similar to the receiver part of Flooding, with the difference that not all intermediate node are allowed to forward data. This restriction can be seen for example at line 37. An intermediate node only forwards a `data_message`, if it is a forwarder node and received a `forwarder_message` with the last part of his RIME address added to the `mprs` attribute.

4.4.3 Directed Diffusion

As discussed in chapter 3.4.4, the *initialization phase* of Directed Diffusion consists of broadcasting `interest_messages` for finding a path from the receiver to the sender. This path is reversed and the data is sent from hop to hop to the receiver via UDP unicast.

Listing 4.7: Sender process of Directed Diffusion

```
1  PROCESS_BEGIN()
2    attach this process to the packet queue
3    send the set_receiver message
4    PROCESS_WAIT for 5 seconds
5
6    while(1){
7      PROCESS_WAIT for packet_event or periodic timer event
8      get the new packet from the packet queue
9
10     if received new data:
11       switch(message_type)
12          case: interest_message:
13            add to the interest cache
14            if "interest collection timer" is over
15               and shortest paths aren't already evaluated:
16
17               evaluate shortest paths
18               setup gradients
19               broadcast reinforcement_message
20               PROCESS_WAIT 10 seconds
21
22               for each fragment i do
23                 PROCESS_WAIT for a random backoff time
24                 buildFragment(i)
25
26                 for each receiver r do:
27                   send fragment i to gradient(r).next_hop
28
```

```
29          case nack_message:
30            resend requested fragemnt for receiver r
31            to gradient(r).next_hop
32          case ack_message:
33            note that receiver finished
34
35     else
36       if all receivers finished:
37         send finished message
38   }
39 PROCESS_END()
```

Listing 4.7 illustrates the pseudo code for the sender part of Directed Diffusion. First the sender broadcasts the `set_receiver_message` (line 3). Then the *while(1)* loop is entered (line 6). This process listens for `packet_events` or `periodic_timer_events` and blocks as long as such an event arrives (line 7). If a new packet arrives, it is treated depending on the message type it represents (line 10-11). The *initialization phase* is started by the receivers. They broadcast `interest_messages` until they receive a `reinforcement_message`. If a `interest_message` arrives, it is added to the interest cache (line 13). If the "interest collection time" is over, then the sender will evaluate the received interests stored in the interest cache (line 14-15). The evaluation of the received interests is illustrated by listing 4.8. When the interests are evaluated and the paths to the receivers are chosen, the sender will setup the gradients (line 18). The gradients for the sender are easy to find: It is just the first hop in the path to the receivers. Next the sender is broadcasting the `reinforcement_message` to initialize the gradient setup of all intermediate nodes on the paths to the receivers (line 19). After some seconds of waiting until all gradients are established, the sender will start to send all fragments one by one (line 22-27). If all fragments were sent, the sender will wait for incoming NACK or ACK messages. If it received an ACK message from all receivers, it will start broadcasting the `finished_message`.

Listing 4.8: Evaluation of the received `interest_messages`

```
1 calcShortestPath(list interest_cache){
2
3 hops_r1 = MAX_HOPS
4 hops_r2 = MAX_HOPS
5 hops_r3 = MAX_HOPS
6
7 found_path_r1
8 found_path_r2
9 found_path_r3
10
11   for each interest I in the cache do:
12
13     for each receiver R in {1,2,3} do:
14
15       if the interest I is from receiver R:
16         hops =  hops of the path from interest I
17
18         if hops < hops_rR
```

```
19            hops_rR = hops
20            found_path_rR = I.path
21
22 for each found_path_ri do:
23   reverse found_path_ri
24 }
```

Listing 4.8 shows the pseudo code for the evaluation of the received interests. The goal of this method is to find and to reverse the shortest path to each receiver. First, this method loops over all `interest_messages` found in the cache (line 10). Then it checks from which receivers the `interest_messages` were sent (line 11-13). To count the number of hops of an `interest_message`, the method only has to loop over the `path` attribute until it finds the value `zero`, because this value is used to mark the end of a path (line 14). For finding the shortest path, the number of hops of `interest I` has to be compared to the variable `hops_rR` (line 15), which was set to the maximum number of possible hops at its initialization (line 3-5). The variable `hops_rR` denotes the corresponding variable for receiver R: hops_r1 for receiver 1, hops_r2 for receiver 2 and hops_r3 for receiver 3. At the first comparison of `hops` and `hops_rR`, `hops` is always lower, since `hops_rR` was set to the maximum of possible hops. Therefore, the value of `hops` is assigned to `hops_rR` (line 16). Also a reference of the corresponding path of the `interest_message I` is assigned to the variable `found_path_rR`. This reference is needed for the case this path should be the shortest (line 17).

If all `interest_messages` were processed, the references to the shortest paths are stored in the variables `found_path_rR`. The only thing which remains to do, is reversing of these paths. That is accomplished by pushing (beginning at the first element) each element of a path to a stack, and then poping each element one by one from the stack again. This procedure returns the reversed paths.

Listing 4.9: Receiver process of Directed Diffusion

```
1  PROCESS_BEGIN()
2  attach this process to the packet queue
3
4  while(1){
5    PROCESS_WAIT for packet_event or periodic timer event
6    get the new packet from the packet queue
7
8      if received new data:
9        switch(message_type)
10          case set_receiver_message:
11             if i'am a receiver:
12                isReceiver=1
13             if ttl > 0: decrement ttl and re-broadcast
14          case interest_message:
15             if the last part of my RIME address is not already
16                on the path:
17                 add it to the path
18                 re-broadcast
19          case reinforcement_message:
20             if not already processed a reinforcement_message:
```

```
21                for each attribute reinf_infos do:
22                   if the last part of my RIME addr is on the path:
23                     setup a gradient to the successor
24                     setup a gradient to the predecessor
25             if ttl > 0: re-broadcast reinforcement_message
26          case data_message:
27                if caching is on: add payload to cache
28                if isReceiver ==0: forward to next hop
29          case nack_message:
30             if requested fragment is in the cache
31               resend fragemnt
32             else
33               forward to next hop
34          case ack_message:
35               if rime_addr_of_receiver == 0: print statistics
36               forward to next hop
37     else
38       if all fragments received and isReceiver == 1
39         appstate=finished
40       else
41         if i'am a receiver and not received a packet for a certain time:
42           request missing fragments
43       if i'am a receiver and appState==finished
44         send ack
45
46    if i'am a receiver and not already received a reinforcement_message:
47      PROCESS_WAIT for 3 seconds
48      send interest
49 }
50 PROCESS_END()
```

Listing 4.9 illustrates the pseudo code for the receiver part of Directed Diffusion. As soon as the receiver process is started, it is broadcasting periodically `interest_messages` until it is receiving a `reinforcement_message` (line 46-48). If an `interest_message` is received, the node first checks if the last part of its RIME address is somewhere on the path of the received message (line 15-16). If so, it drops the message, because this `interest_message` was already forwarded by this node. By dropping this `interest_message`, the node can avoid loops in the path from the sender to the receiver.

If this message was not already forwarded by this node, the node adds the last part of its RIME address to the next free place in the `path` attribute and rebroadcasts the message (line 17-18). After a certain time, the sender process will broadcast the `reinforcement_message`. If a node receives this message, it will be processed only once (line 20). Since this `reinforcement_message` consists of three `reinf_infos` attributes, the node has to iterate over these three attributes and to search whether the last part of its RIME address is somewhere on the path (line 21-22). If there is a match, the node is on the path from the sender to a receiver, and it has to forward data coming from the sender towards the receiver, and backwards from the direction of the receiver to the sender. To be able to forward this data, the gradients have to be established for the receiver specified in the corresponding `reinf_infos` attribute (line 23-24). If the TTL value of the received `reinforcement_message` is greater than zero, it

will be re-broadcasted (line 25). If a `data_message` arrives, it will be directly forwarded to the next hop (line 28).

Chapter 5

Evaluation

5.1 External Factors

As discussed in chapter 1.2, the three implemented communication protocols were evaluated in a real sensor network testbed. During the evaluation of these protocols, we had to cope with well known problems and limitations of radio communication: External factors like high voltage power lines, Bluetooth and other interferer directly influence the quality of the radio communication. *We observed these limitations through the fact, that during the day the evaluated protocols without any other reason did not work in the chosen testbed.*

The effect of the above mentioned factors led to a massively increased packet loss, and thus to an extremely slow down of the data communication. Since the experiments have a limited execution time, they will not finish if the packet loss rate (PLR) is too high, and therefore fail. We tried to avoid these problems by scheduling the experiments always at midnight.

5.2 Testbed Topologies

The used testbed consists of 40 TmoteSky sensor nodes. To get an idea of the connectivity situation between the different sensor nodes, the link between two sensor nodes was tested by sending a certain amount of UDP unicast packets and counting how much of them arrived. If more than 95% of the packets arrived, this link was classified as good. Figure 5.1 shows the TARWIS testbed.

The green lines indicate good links, meaning that at least 95 % of all sent packets arrived. Based on this connectivity situation, *node 7* was chosen as sender node, and the *nodes 16,19, and 24* as receiver nodes. This choice enables at least a three hop scenario, meaning that the packets have to be forwarded at least three times.

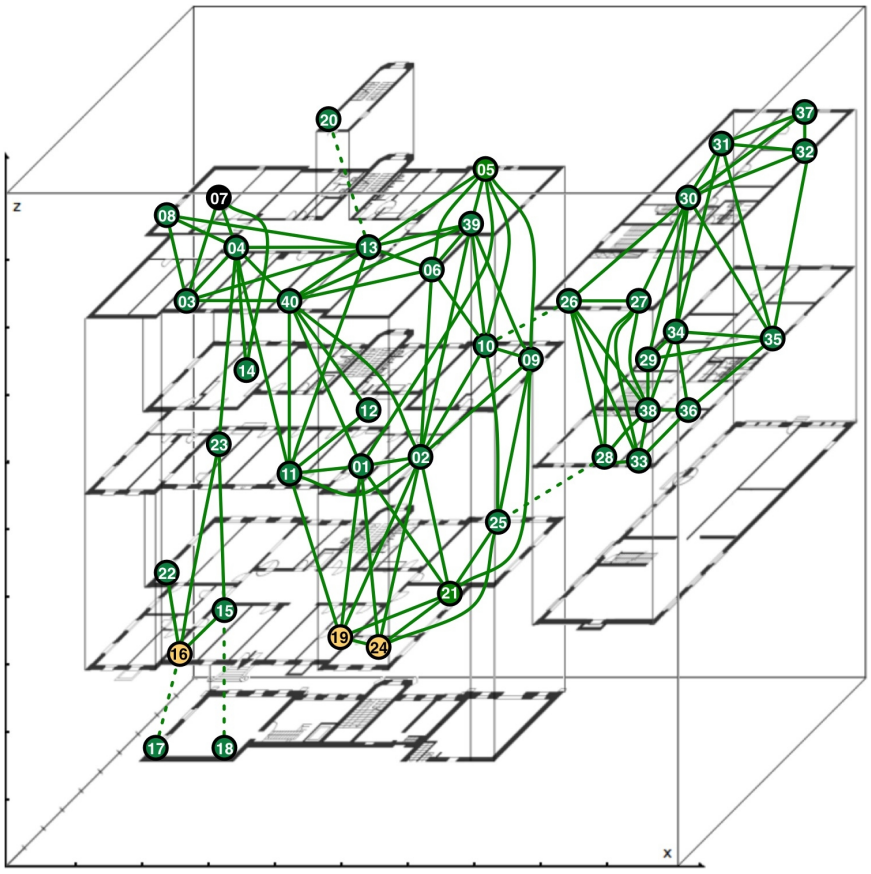

Figure 5.1: BigNet scenario

For evaluating the implemented communication protocols, two different scenarios were set up: The first scenario is denoted as *BigNet*, because in this scenario all 40 sensor nodes of the testbed are used. In the second scenario only 9 sensor nodes are used, and therefore it is denoted as *SmallNet*.

This separation was necessary, because we were not able to evaluate the implemented protocols with `ContikiMAC` with all 40 sensor nodes of the testbed. Using all 40 sensor nodes and `ContikiMAC`, data transmission was very slow and exceeded the defined maximum experiment time of 30 minutes. Whereas with 9 instead of 40 sensor nodes we were able to evaluate Flooding with caching and UDP unicast with `ContikiMAC`, and therefore we had to setup two different scenarios.

Figure 5.2 shows the *SmallNet* scenario. The *SmallNet* scenario consist only of the nodes 1, 4, 7, 11, 16, 19, 23, 24, 40.

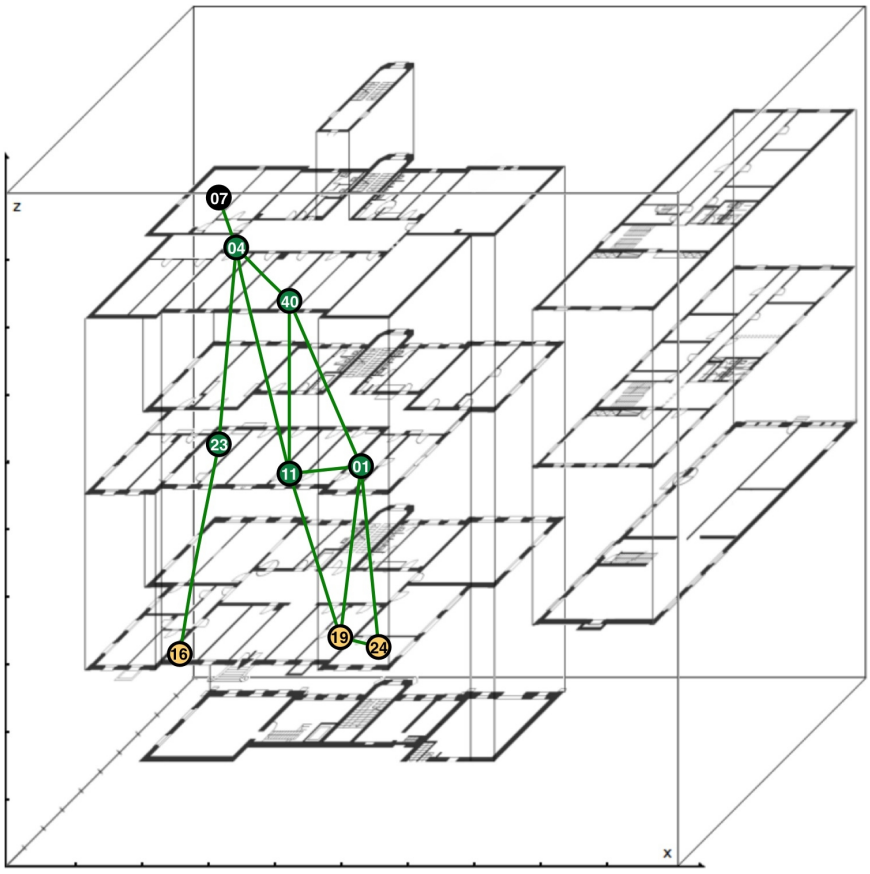

Figure 5.2: SmallNet scenario

5.3 Evaluation Metrics

5.3.1 Transmission Time

The first evaluation metric is the transmission time of the chosen payload. The measurement of the transmission time starts with the sending of the first fragment and ends with the reception of the last ACK message. Therefore, the transmission time is equal to the time used for the *send data phase*. The time used for the *initialization phase* is not considered. For measuring the time between these two events, *first fragment sent* and *last ACK received*, the sender sensor node prints a distinct string, when one of these events happens. As mentioned in chapter 3.4.1, every sensor node in the used testbed is attached to a corresponding mesh node. Contiki modified the common `print` statement of C in that way, that every string supplied as function argument is sent via RS232 to the attached mesh node. TARWIS collects and timestamps all output gathered at every mesh node and provides a XML file with tagged data.

For measuring the transmission time, the sender prints the string `SENDER: TIME_START` before sending the first fragment and `SENDER: TIME_STOP` after the reception of the last ACK message. After the successful TARWIS experiment, the corresponding XML file can be

analyzed. Since all output was timestamped by TARWIS, the difference between the timestamp for `SENDER: TIME_STOP` and `SENDER: TIME_START` can be calculated, which leads to the transmission time. It is obvious that this measurement method is not absolutely precise, since it takes time to transmit the above mentioned strings via RS232 to the attached mesh node. Besides this small error, the much bigger error results from the fact, that each receiver has to send an ACK message to the receiver to signal successful data transmission. It takes time until this ACK message arrives at the sender.

Since the gathered results depend on the specific topology used, not the actual transmission time is of value, but instead the relation between the different transmission times of the different evaluated protocols. By using the same measurement method for all evaluated protocols, the small bias for the transmission time is equal for all evaluated protocols.

5.3.2 Energy

The third evaluation metric is the used energy of all participating sensor nodes for transmitting the payload. A software based energy measurement method is used as discussed in chapter 2.3.5. As with the sent frames and collisions, the energy measurement starts with the *send data phase* and does not consider the *initialization phase*. All intermediate nodes print the summed up *energest* values after the reception a corresponding ACK message. Through an offline analysis of the corresponding XML file, we are able to calculate the consumed energy by simply summing up the *energest* values for listening and transmitting and then converting these values to Joule as explained in chapter 2.3.5.

5.3.3 Sent Frames and Collisions

The second evaluation metric is the number of sent frames and the number of collisions during the *send data phase* for all participating sensor nodes. For counting the number of sent frames and the number of collisions, the `mac_call_sent_callback` interface provided by Contiki was used. Each MAC protocol has to implement this interface, and therefore it is a generic way to count frames and collisions, which works for all supported MAC protocols.

Listing 5.1: `mac_call_sent_callback` interface of Contiki OS

```
1  unsigned int macCollision=0;
2  unsigned int macSent=0;
3
4  void
5  mac_call_sent_callback
6  (mac_callback_t sent, void *ptr, int status, int num_tx){
7
8    switch(status) {
9    case MAC_TX_COLLISION:
10     PRINTF("mac: collision after %d tx\n", num_tx);
11     macCollision++;
12     break;
13   case MAC_TX_NOACK:
14     PRINTF("mac: noack after %d tx\n", num_tx);
15     break;
```

```
16    case MAC_TX_OK:
17      PRINTF("mac: sent after %d tx\n", num_tx);
18      macSent++;
19      break;
20    default:
21      PRINTF("mac: error %d after %d tx\n", status, num_tx);
22    }
23    if(sent) {
24      sent(ptr, status, num_tx);
25    }
26  }
```

Listing 5.1 shows the used `mac_call_sent_callback` interface. For counting the sent frames and the collisions, two global variables were defined, which hold the counted values (line 1-3). An increment statement was added at the necessary positions: For counting the sent frames, the `macSent` variable is incremented (line 20). For counting all the collisions, the `macCollision` variable is incremented (line 12).

Before sending the first fragment, these two variables are set to zero, since the sent frames and collisions during the *initialization phase* are not counted. If the data transmission succeeded, all intermediate nodes will receive an ACK message with the attribute `rime_addr_of_receiver` set to zero. In this case the intermediate nodes will print the value of two above mentioned variables. Through analyzing the corresponding XML file, we are able to sum up all sent frames and collisions for the whole WSN.

5.4 Evaluation Procedure

As discussed in chapter 5.3.1, TARWIS provides for every executed experiment a XML file with all gathered output from the used sensor nodes. To get the data necessary for the evaluation, a *two-pass parser* was implemented, which filters out the wanted data from the XML file and stores it to the `SQlite` [21] database for further processing. The first pass of the parser eliminates all unused XML tags and returns only the data printed by the sensor nodes in the format `timestamp [TAB] node_id [TAB] printed string`. The second pass of the parser picks out all lines corresponding to a certain `node_id` and and searches for the wanted data using *regular expressions*. The gathered data is stored to a `SQlite` database, which is a very lightweight *file-based* database [21].

As discussed in chapter 1.2, the implemented protocols are evaluated with different payload sizes, caching strategies and MAC protocols. This adds up to many different configurations, which have to be evaluated in the testbed. To keep track of the progress, a simple script called `makePlotData` was implemented, which displays the content of the `SQlite` database in a table and produces GNU Plot [22] boxplot-files for a quick overview of the gained results.

Listing 5.2: Output of the `makePlotData` script

```
1 .-------------------------------------------------+------------+------------.
2 | scenario                                        | number of- | status     |
3 |                                                 |  records   |            |
4 '-------------------------------------------------+------------+------------'
5 .-------------------------------------------------+------------+------------.
```

```
| found udp_unicast_no_cache_nullmac_nullrdc_1000    | 23         | OK         |
+----------------------------------------------------+------------+------------+
| found udp_unicast_cache_CSMA_ContikiMAC_1000       | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found udp_unicast_cache_nullmac_nullrdc_1000       | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found flooding_no_cache_nullmac_nullrdc_1000       | 24         | OK         |
+----------------------------------------------------+------------+------------+
| found flooding_cache_nullmac_nullrdc_1000          | 23         | OK         |
+----------------------------------------------------+------------+------------+
| found flooding_pro_active_nullmac_nullrdc_1000     | 32         | OK         |
+----------------------------------------------------+------------+------------+
| found flooding_cache_nullmac_ContikiMAC_1000       | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found mpr_no_cache_nullmac_nullrdc_1000            | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found mpr_cache_nullmac_nullrdc_1000               | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found mpr_pro_active_nullmac_nullrdc_1000          | 20         | OK         |
+----------------------------------------------------+------------+------------+
| found dd_cache_CSMA_nullrdc_1000                   | 25         | OK         |
+----------------------------------------------------+------------+------------+
| found udp_unicast_no_cache_nullmac_nullrdc_70      | 24         | OK         |
+----------------------------------------------------+------------+------------+
| found udp_unicast_no_cache_CSMA_ContikiMAC_70      | 21         | OK         |
+----------------------------------------------------+------------+------------+
| found udp_unicast_cache_nullmac_nullrdc_70         | 26         | OK         |
+----------------------------------------------------+------------+------------+
```

Listing 5.2 shows the output of the `makePlotData` script. This script searches the `SQlite` database for all different configurations of payload size, caching strategy and MAC protocol and prints each found configuration and the number of corresponding experiments. To have representative data, we decided to run at least 20 experiments for every configuration. The last column of the output of the `makePlotData` script shows the status of the corresponding configuration. If more than 20 experiments were found, the status shows "OK", otherwise "!!".

For further processing of the experiment data stored in the `SQlite` database, we used the `GNU R` [23] statistical environment. `GNU R` provides a `SQlite` Database Inteface (DBI), which makes it possible to import the experiment data to `GNU R`. Since the results do not have a Gaussian distribution, boxplot diagrams are used to visualize the distribution of the data.

5.4.1 Reference Protocol

Besides the discussed communication protocols, the data transmission using pure UDP unicast was also implemented. The data gathered from the experiments with this protocol provides suitable reference data for comparing the other protocols. This protocol uses the same NACK-based reliability mechanism as described in chapter 3.2. Caching support is implemented the same way as found by Directed Diffusion. A closer look at these two protocols shows, that the differences are not big. In fact, the *send data phases* of the two protocols are equal. The differences are found in the *initialization phase*. UDP unicast does not use any *initialization phase* at all, since it uses hard coded and pre-configured routing tables. Directed Diffusion has to use the *initialization phase* to discover a path from the sender to the receiver and to setup gradients. After these preparations, Directed Diffusion transmits the data using UDP unicast to the next hop. Our reference protocol therefore represents the best possible case of Directed Diffusion: If Directed Diffusion would discover the best possible paths, it would perform equally as UDP unicast.

UDP unicast without any caching is implemented differently as Directed Diffusion without any caching. Directed Diffusion is in fact very different from IP-style communication, since it

has to use the gradients for forwarding the data to the next hop. Therefore it does not use IP forwarding. The whole protocol is located at the application layer, also the forwarding of the data. A packet has always to travel up the whole protocol stack until it reaches the application layer, despite it is not directly sent to this particular node.

That is different with UDP unicast without any caching. Since this protocol uses pre-configured routing tables and is not implemented at the application layer (therefore it does not support caching), IP forwarding can be used. That means a packet is addressed directly to the receiver, and is forwarded by the intermediate nodes at the network layer and *not* at the application layer. A packet has therefore not to travel up the whole protocol stack until it reaches the application layer to be forwarded.

5.5 Results

As discussed in chapter 5.2, two different scenarios are used due to the fact, that the implemented protocols did not work with `ContikiMAC` in the *BigNet* scenario. Therefore, we present first the results for the *BigNet* scenario and afterwards the results for the *SmallNet* scenario.

5.5.1 Transmission Time

Figure 5.3: Transmission time for 1000 bytes in the BigNet scenario using `NullMAC`

BigNet Scenario

Figure 5.3 shows the results of transmission time of a payload size of 1000 bytes using `NullMAC`. The black dots, which can be seen on top or on bottom of some boxplots symbolize outliers. An outlier is any value that lies more than one and a half times the length of

the box from either end of the box. The first column shows the results for Flooding using different caching strategies. The second column shows the results for MPR, and the third column shows the results of UDP unicast. It can be seen, as expected, that the caching strategy highly influences the time used for transmitting the payload. Without any caching, the NACK messages from the receiver have to travel back to the sender, until the desired fragment are resent. Caching the fragments on every intermediate node therefore significantly reduces the time until a requested fragment arrives, since the NACK message can be replied by any intermediate node, which cached the fragment.

Flooding

Flooding with a proactive caching strategy seems to be as fast as Flooding with normal caching, but that is not the whole story. First, we implemented proactive caching in that way, that every intermediate node requested fragments as soon it did not receive a packet for a certain interval. This resulted in a huge traffic consisting of NACK messages, and therefore produced a lot of collisions. A lot of packets were lost and the data transmission did not finish within an acceptable time. We therefore introduced a *request limit*, which allows the intermediate nodes to request only a certain amount of fragments. If the *request limit* is reached, the intermediate nodes will act similarly as with regular caching. With these modifications, Flooding with proactive caching is slightly faster as Flooding with regular caching, because intermediate nodes are more often able to reply to NACK messages. The median for the transmission time for Flooding with proactive caching is 84.34 seconds, for Flooding with regular caching 89.80 seconds.

The wide variance of the transmission time of Flooding with regular caching is noticeable. 50% of the transmission times are in the interval [86.89, 173.96]. That is a very wide interval, especially compared the other protocol configurations. We assume that this wide variance results from the fact, that without any caching the traffic in the WSN increases massively. This traffic is amplified again through the "flooding" of fragments through the WSN: Every intermediate node re-broadcasts the received messages. This leads to a very non-deterministic situation, where a lot of collisions occur and a lot of packets are lost, which results in a big variance.

MPR

The benefit of MPR results in a shorter transmission time for all caching strategies compared to Flooding. Even the transmission time for MPR without any caching is shorter than the transmission time for Flooding with regular caching. It is surprising that the transmission time of MPR with regular caching is almost equal to MPR without any caching. We would expect that MPR benefits from regular caching as Flooding does. We assume that the data gathered for MPR with regular caching was biased through external factors as discussed in chapter 5.1.

Directed Diffusion

The third communication protocol to evaluate in the *BigNet* scenario would be Directed Diffusion, but it did not work properly in this scenario. The main problem of Directed Diffusion was broadcasting `interest_messages`. The *BigNet* scenario consists of 40 sensor nodes, and the

path of the broadcasted `interest_messages` towards the sender can not be influenced. The resulted paths from the sender towards the receivers were very long and mostly consisted of 7 to 10 hops, due to broadcasting of the `interest_messages`. The optimal paths are about 3-4 hops. Besides the too longs paths, the quality of hop-by-hop links of the found paths often was bad. The fact that the `interest_message` was forwarded once from one node to an other does not say anything about the link quality. Often the link quality between two hops was very poor, because when broadcasting a packet, each node within the transmission range receives it. This often lead to hop-by-hop links between nodes with a large distance in between, and therefore mostly to unreliable links. This combination, long paths and unreliable hop-by-hop links, are the crucial factors for failure of Directed Diffusion in the *BigNet* scenario. To proof our assumption, we also evaluated Directed Diffusion in the *SmallNet* scenario.

SmallNet Scenario

As Figure 5.4 shows, with less sensor nodes and an advantageous selection of the position of these nodes, Directed Diffusion finally worked. This Figure also shows Flooding with regular caching and `ContikiMAC` as used MAC protocol, and for reference, UDP unicast with regular caching running on top of `NullMAC` and `ContikiMAC`.

Comparing the medians for the transmission time of Flooding with regular caching using `ContikiMAC` on one hand and `NullMAC` on the other hand, the slow down of the transmission time by using `ContikiMAC` can be observed. The median using `NullMAC` is 89.28 seconds, using `ContikiMAC` it is 188.35 seconds. It can be seen that the RDC mechanism of `ContikiMAC` is responsible for a doubled transmission time.

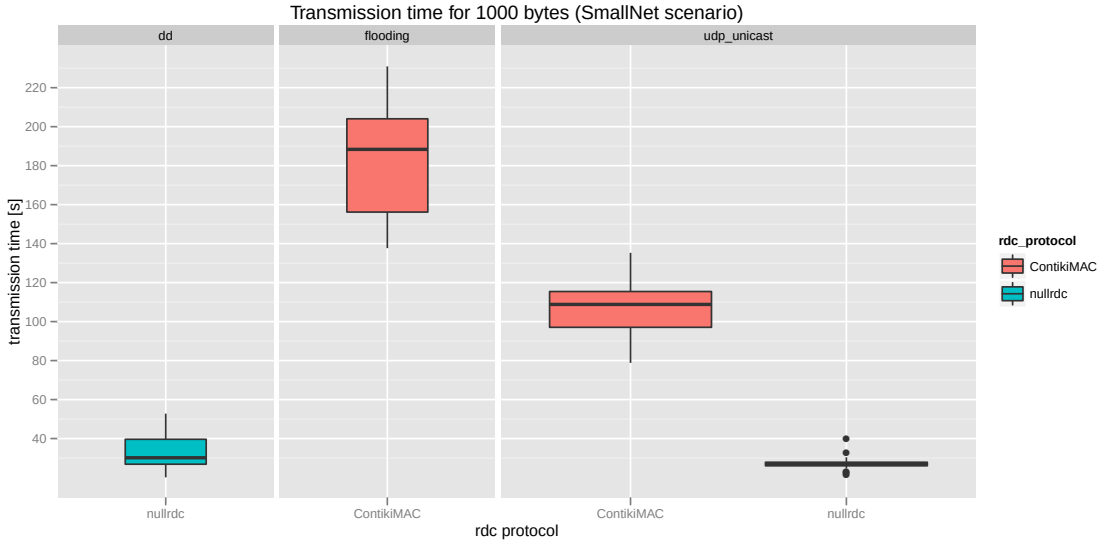

Figure 5.4: Transmission time for 1000 bytes with caching in the SmallNet scenario

The fact that Directed Diffusion works in the *SmallNet* scenario proofs the assumption made above, since the *SmallNet* scenario consists only of 9 sensor nodes and the length and the direction of the paths is restricted. The time to transmit 1000 bytes using Directed Diffusion with regular caching is quite good, the median is 30.14 seconds. The median for the transmission time of UDP unicast with regular caching is 26.99 seconds. This indicates that despite the restricted possibilities to choose a path, Directed Diffusion still does not find the optimal paths. UDP unicast uses the optimal paths, since they are hard-coded and preconfigured.

The difference for the transmission time between UDP unicast with regular caching using `ContikiMAC` on one hand and `NullMAC` on the other hand is quite huge. The median using `ContikiMAC` is 109.12 seconds, and using `NullMAC` 26.99 seconds. This shows that the data transmission using `ContikiMAC` is about four times slower compared of using `NullMAC`. But `ContikiMAC` also has its advantages: The used RDC mechanism is able to save a lot of energy. Taking the data of UDP unicast as reference to compare with the other protocols shows that a usual UDP connection is much faster than Flooding or MPR. But this comparison is not fair, since UDP unicast depends on hard-coded routing tables, whereas Flooding and MPR do not.

Figure 5.5 shows the results for the transmission time for a payload size of 70 bytes. It can be seen that for one single packet (70 bytes payload), the difference between the transmission times are not as big as for 15 packets (1000 bytes payload). Just Flooding without any caching takes a little bit longer. Obviously for a payload fitting into one packet, the caching strategies does not matter any more, and therefore the costs for caching are too high in relation for the benefit.

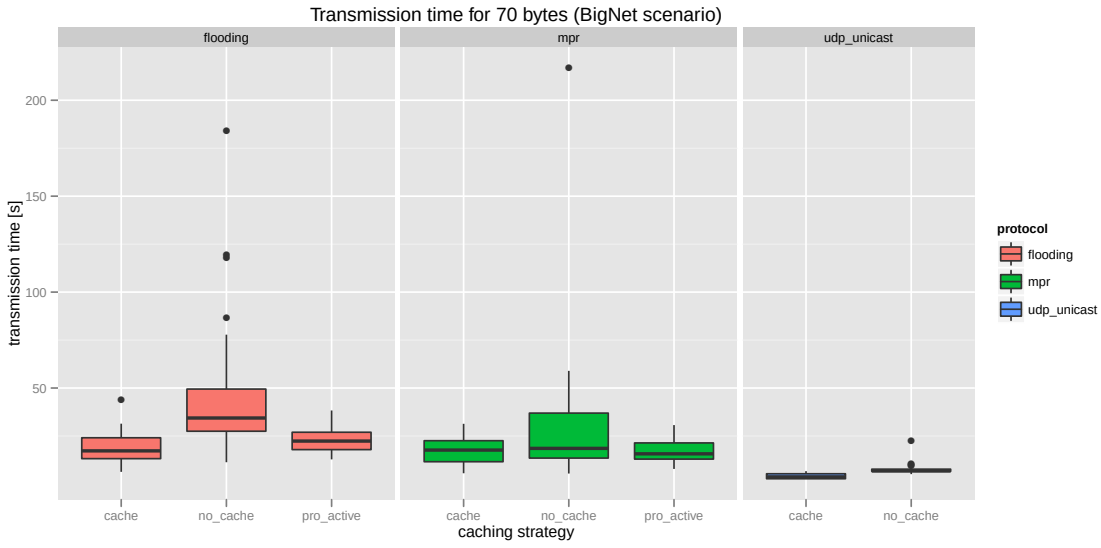

Figure 5.5: Transmission time for 70 bytes in the BigNet scenario

5.5.2 Energy

BigNet Scenario

Figure 5.6 shows results for the energy consumption for transmitting 1000 bytes in the *BigNet* scenario.

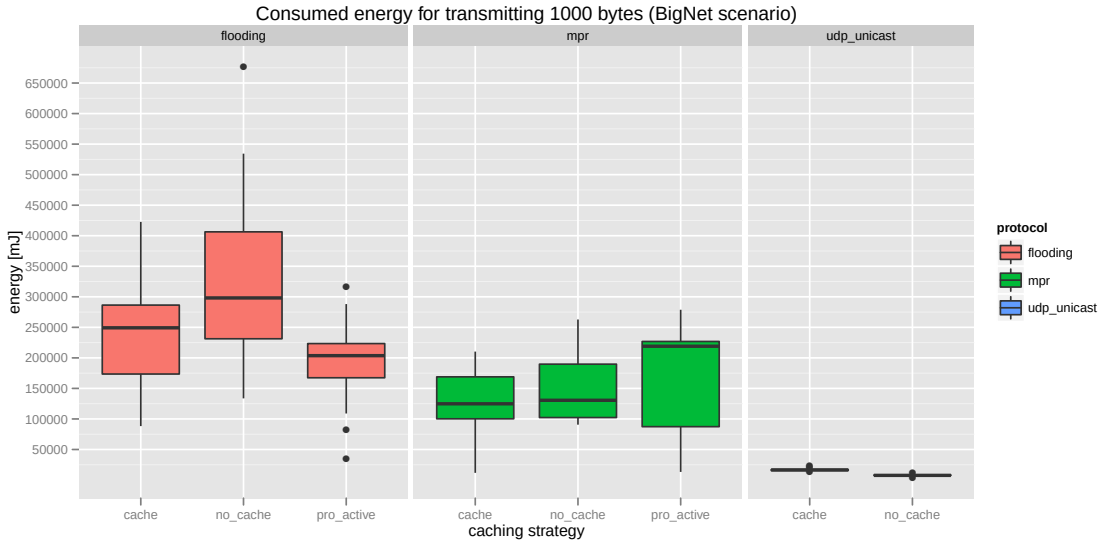

Figure 5.6: Consumed energy for transmitting 1000 bytes

Since in the *BigNet* scenario only `NullMAC` is used, the energy consumption should be *linear* to the transmission time.

Checking the linearity between the transmission time and the energy consumption shows that the expectation made above comes true, except for MPR with proactive caching. A lower value for the energy consumption is expected, since the median of the transmission time of MPR with proactive caching is the lowest of all caching strategies. Therefore, the energy consumption has to be the lowest, since without any radio duty cycling the consumed energy has to be proportional to its transmission time. Obviously the energy data for MPR with proactive caching was biased during its measurement.

Figure 5.7 shows the consumed energy plotted against the transmission time of MPR with proactive caching using `NullMAC`.

A straight line is expected, *due to the linear correlation between the consumed energy and the transmission time.* We can spot that the first records satisfy linearity, until a big valley appears. This record is obviously a measuring error. The records afterwards satisfy linearity again, until the straight line falls again. Obviously these records now are biased, since it is impossible that transmitting data during 100 seconds uses less energy as transmitting data during 50 seconds.

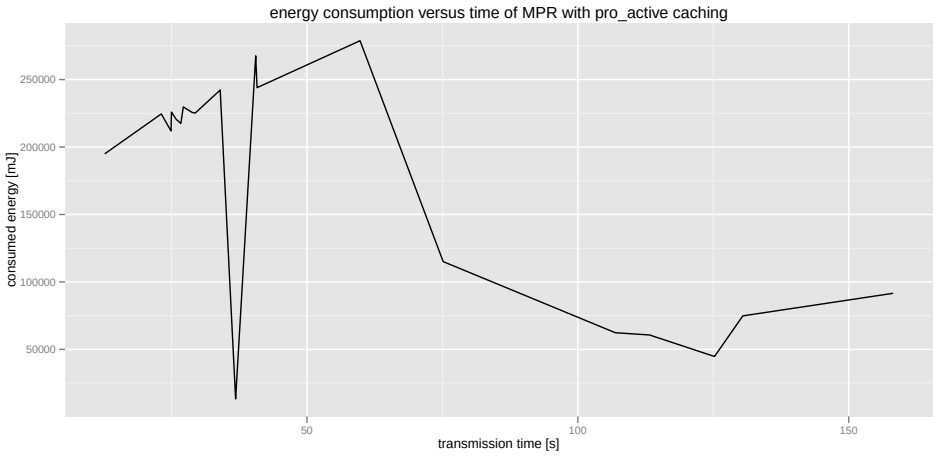

Figure 5.7: The biased data of MPR with proactive caching

We assume that the proactive caching strategy disturbs the reception of the `finished_message`, and therefore the printing of the *energest* values. That is possible if a lot of intermediate nodes did not reach the *request limit* until the sending of the `finished_message`. This leads to the following situation: The actual data transmission finished, but the intermediate nodes are still requesting fragments, until they reach the *request limit*. These attempts to request fragments disturb the reception of the `finished_message` and are therefore responsible for the biasing of the data.

Quality Checking of the Data

After the discovery of the biased data, the other data was also analyzed regarding the correlation between transmission time and energy consumption. This was done by calculating the Spearman's rank correlation coefficients [24] for all configuration of Flooding, MPR and UDP unicast. These correlation coefficients show the correlation between transmission time and used energy expressed as a decimal, which lies in the interval [-1,1]. A correlation coefficient with the value 1 indicates absolute linearity, whereas a correlation coefficient with the value -1 indicates no linearity between transmission time and used energy.

Table 5.1 shows the correlation coefficients for the transmission time and the consumed energy for Flooding, MPR and UDP unicast. It can be seen that the proposition about the linearity of the transmission time and the consumed energy comes true, although some values differ a little bit from the optimum. *As we have already seen, the data for MPR with proactive caching is biased, as the correlation coefficient of -0.35 for this data shows.*

	cor(time, energy)
Flooding cache	0.78
Flooding nocache	0.9
Flooding proactive	0.61
MPR cache	0.96
MPR nocache	0.98
MPR proactive	-0.35
UDP unicast cache	0.97
UDP unicast nocache	0.92

Table 5.1: Spearman's rank correlation coefficients for all configurations of Flooding, MPR and UDP unicast for a payload size of 1000 bytes

The other value, which is too low compared to the other correlation coefficients, is the value of Flooding with proactive caching. That is not really a surprise, since there the same problems arise as with MPR with proactive caching, due to the proactive caching mechanism. Since we know now for sure, that the consumed energy is linearly dependent on the transmission time, a simple linear model can be set up to calculate the measurement error of the gathered data used to calculate the energy consumption.

Figure 5.8 shows a scatterplot of the consumed energy of Flooding with caching.

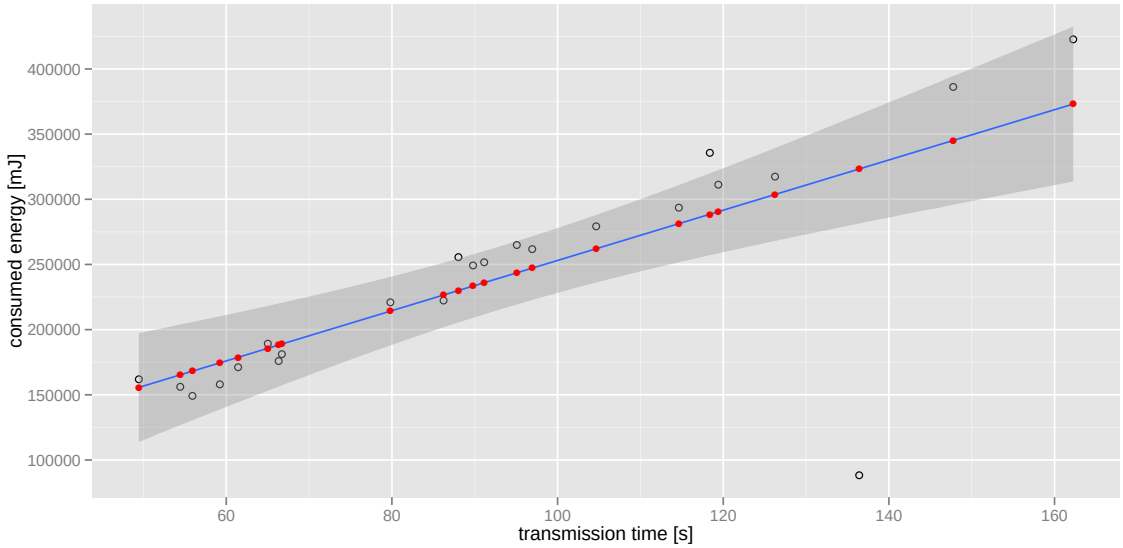

Figure 5.8: Linear regression for the consumed energy of Flooding with caching

The blue line represents a simple linear model developed by applying linear regression to the energy data. The red dots are the calculated values for the energy consumption using this linear model. The dark grey area denotes the 95% confidence interval. The linear model for the consumed energy of Flooding with caching can be described as

$$e_{flood\_1000} = 60369 + 1927 * t_{flood\_1000} \tag{5.1}$$

The median for the consumed energy of Flooding with caching is 249'241.1 mJ. The median calculated with the linear model is 239'273.8 mJ with a probability of 95 %. Therefore, the relative error is about 4%, which is acceptable.

Figure 5.9 shows the energy consumption for the transmission of 70 bytes in the *BigNet* scenario.

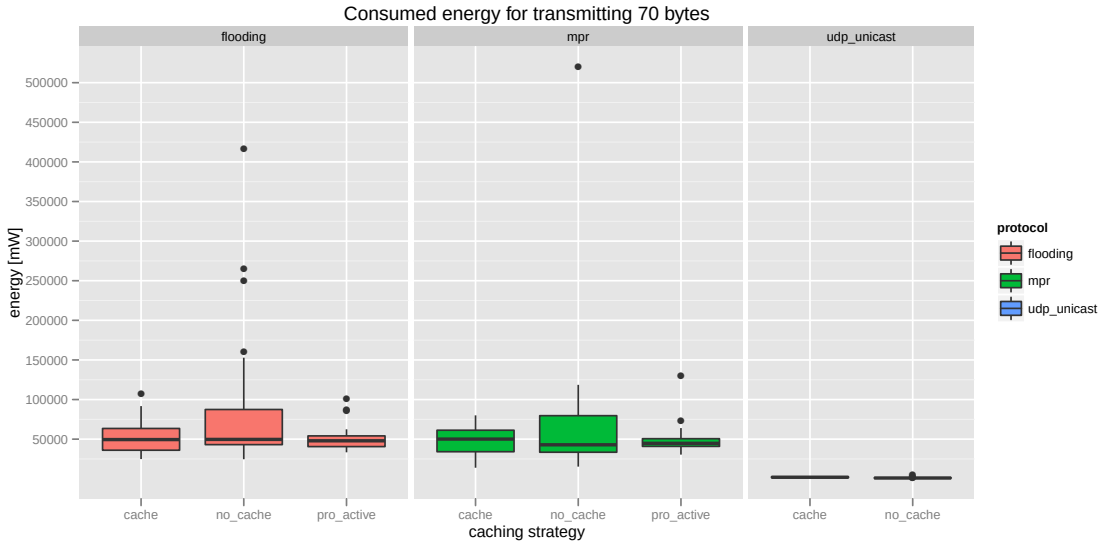

Figure 5.9: Consumed energy for transmitting 70 bytes

As done previously, the Spearman's rank correlation coefficients were calculated for all configuration of Flooding, MPR and UDP unicast, but this time for a payload size of 70 bytes. Table 5.2 shows the results.

	cor(time, energy)
Flooding cache	0.95
Flooding nocache	0.9
Flooding proactive	0.76
MPR cache	0.97
MPR nocache	0.95
MPR proactive	0.7
UDP unicast cache	0.95
UDP unicast nocache	0.88

Table 5.2: Spearman's rank correlation coefficients for all configurations of Flooding, MPR and UDP unicast for a payload size of 70 bytes

It can be seen that the lowest correlation coefficients correspond to Flooding with *proactive* caching and MPR with *proactive* caching. There seems to be definitively a problem with proactive caching and the gathering of the *energest* values. The rest of the data seems to be of a good quality.

As expected, it can be seen that the differences for the energy consumption for transmitting one packet is marginal for Flooding and MPR.

SmallNet Scenario

Figure 5.10 shows the consumed energy for transmitting 1000 bytes in the *SmallNet* scenario.

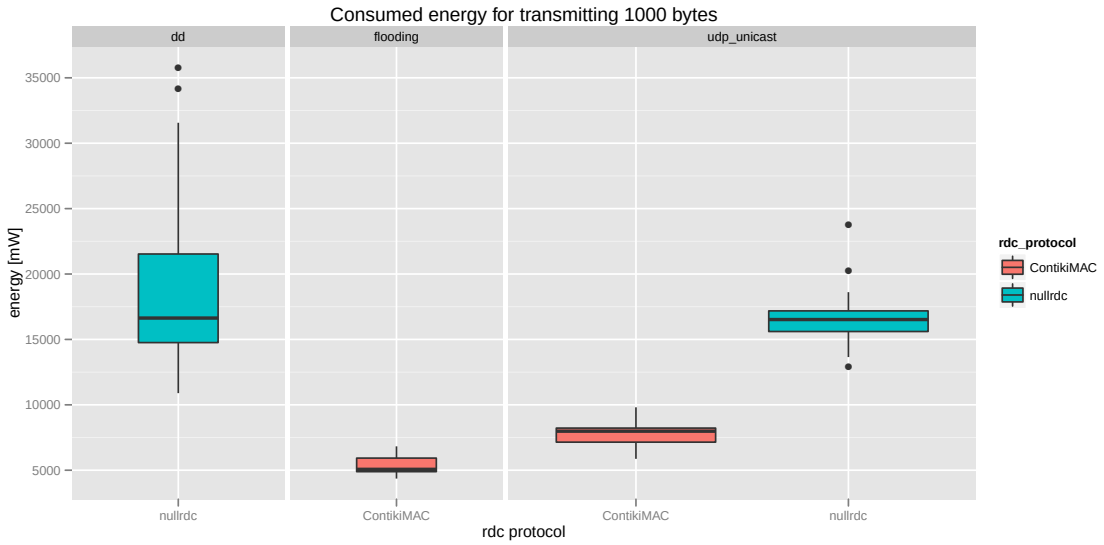

Figure 5.10: Consumed energy for transmitting 1000 bytes in the SmallNet scenario

It is noticeable that the consumed energy of Flooding is lower than the consumed energy of UDP unicast. We would expect that it should be the other way around, since the transmission time for Flooding is much higher than for UDP unicast. To check the quality of the data, the Spearman's rank correlation coefficients were calculated again.

Table 5.3 shows the calculated correlation coefficients.

	cor(time, energy)
DD cache, nullrdc	0.93
Flooding cache, contikimac	0.89
UDP unicast, contikimac	0.82
UDP unicast, nullrdc	0.97

Table 5.3: Spearman's rank correlation coefficients for all configurations of the *SmallNet* scenario

It can be seen that the quality of the data seems to be fine regarding the linearity of the transmission time and the consumed energy. Therefore it is surprising, that Flooding consumed less energy than UDP unicast. We assume that the energy data of Flooding was biased otherwise, because the energy consumption of Flooding should at least be as high as the energy consumption of UDP unicast.

The difference of the energy consumption of UDP unicast using `ContikiMAC` and UDP unicast using `NullMAC` is remarkable. This difference results from the used RDC mechanism of `ContikiMAC`. The median of the consumed energy using `ContikiMAC` is 7988.87 mJ, using `NullMAC` it is 16519.56 mJ. The saved energy by using `ContikiMAC` is about 51.64 %. Putting this result into relation to the transmission time, it can be observed that for the case of UDP unicast, `ContikiMAC` is able to save about 50% of the energy with the costs of a four times slower data transmission.

5.5.3 Sent Frames and Collisions

BigNet Scenario

Figure 5.11 shows the number of sent frames for transmitting 1000 bytes in the *BigNet* scenario. The huge number of sent frames using Flooding or MPR can be observed. The median for Flooding with caching is 48821, the median for MPR with caching is 13976.5. This shows the benefit of reducing the number of forwarding intermediate nodes, because this reduces the number of collisions, and therefore also the number of sent frames.

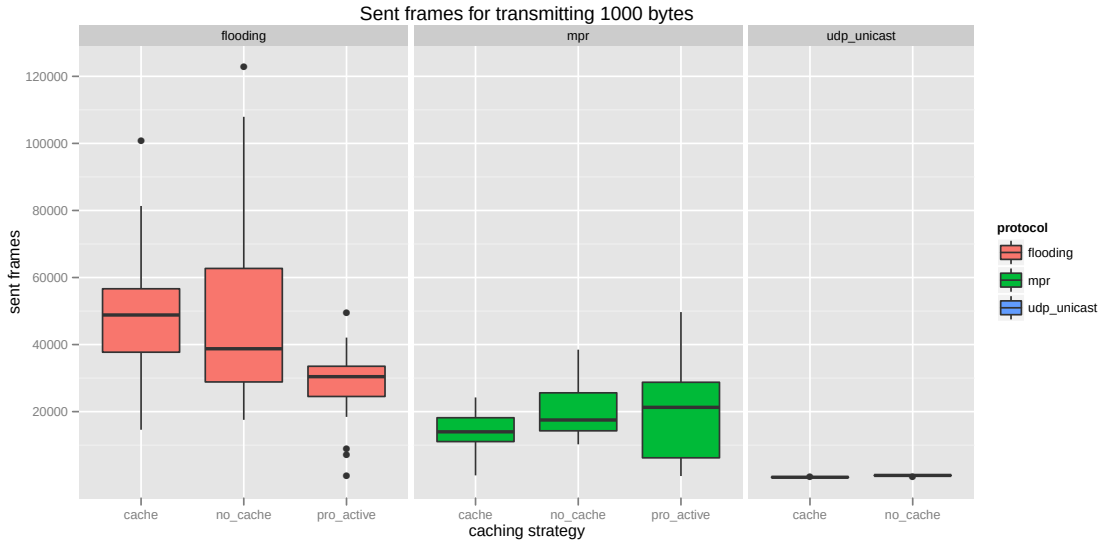

Figure 5.11: Sent frames for transmitting 1000 bytes in the *BigNet* scenario

Figure 5.12 shows the counted collisions while transmitting 1000 bytes. We see the reduced number of collisions produced by MPR compared with pure Flooding. Also, we can see that

proactive caching produces a lot of additional collisions. This results from the fact, that every intermediate node is requesting additionally fragments by itself.

Comparing the number of sent frames and collisions of Flooding, MPR and UDP unicast shows the difference between broadcast-based and unicast-based communication protocols. Broadcast-based communication results in many collisions, due to that every fragment is rebroadcasted many times. However, unicast-based communication produces considerably less collisions, since the fragments are resent only by one intermediate node.

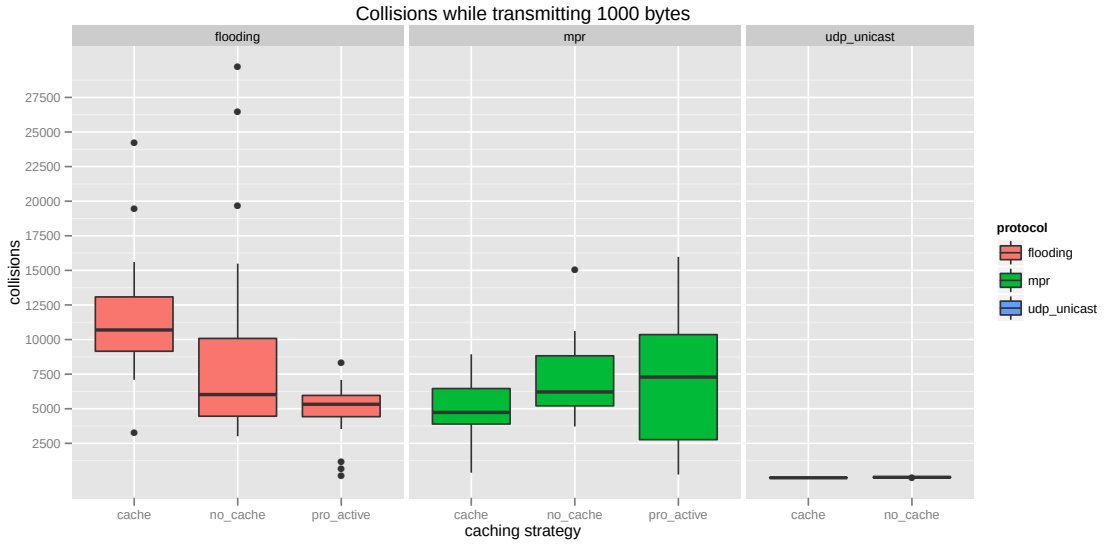

Figure 5.12: Collisions while transmitting 1000 bytes *BigNet* scenario

Figure 5.13 and 5.14 show the number of sent frames and collisions for transmitting 70 bytes in the *BigNet* scenario. It can be observed that for a payload size of 70 bytes, the difference of sent frames and collisions for the several protocols becomes smaller. It is noticeable that the regular caching mechanism seems to result in more sent frames and collisions compared of using no or a proactive caching mechanism.

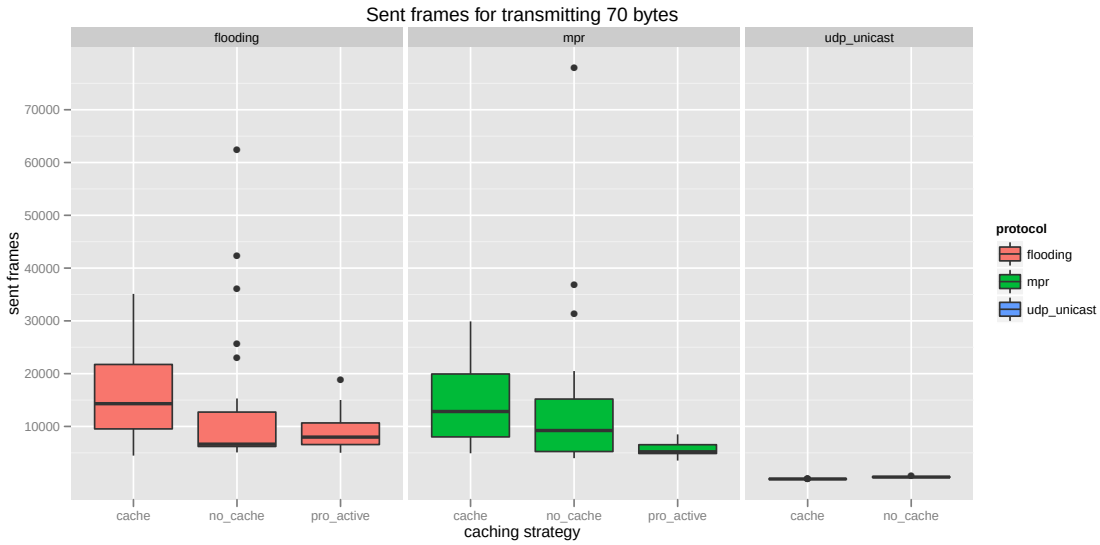

Figure 5.13: Sent frames for transmitting 70 bytes in the *BigNet* scenario

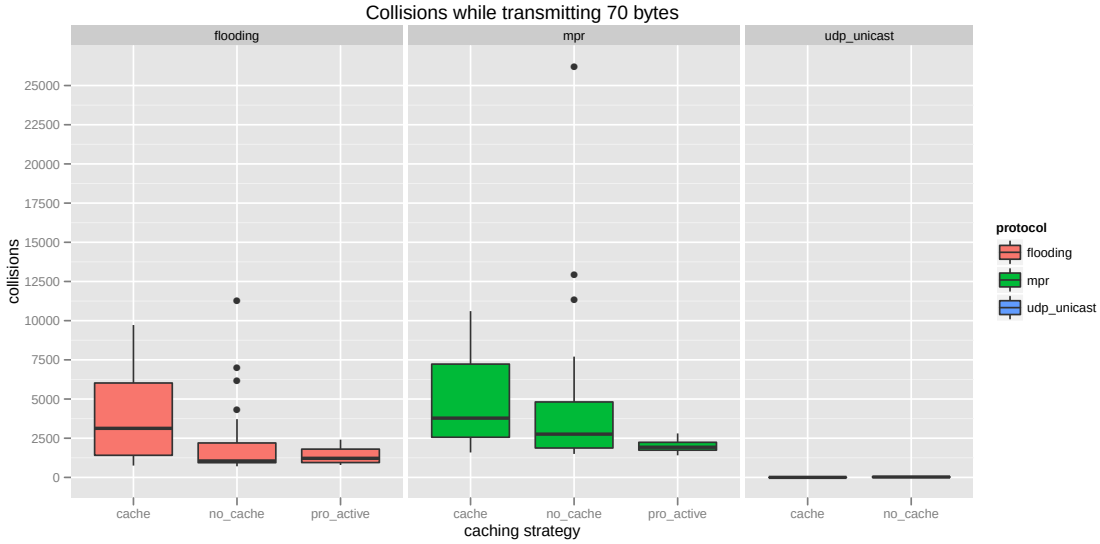

Figure 5.14: Collisions while transmitting 70 bytes in the *BigNet* scenario

SmallNet Scenario

Figure 5.15 and 5.16 show the number of sent frames and collisions for transmitting 1000 bytes in the *SmallNet* scenario. It can be seen that the usage of `ContikiMAC` results in many additional collisions. An other evidence for the shortcomings of the path discovery of Directed

Diffusion can be discovered: As it can be seen, the number of collisions of Directed Diffusion are in the same magnitude as of UDP unicast using `NullMAC`. But the number of sent frames is much higher as for UDP unicast using `NullMAC`.

That is an evidence for choosing too long and bad paths by Directed Diffusion, because the fragment is sent along this long path, which results in a increased number of sent frames. The number of collisions of Flooding using `ContikiMAC` is also quite high compared to the number of sent frames. This high number of collisions using `ContikiMAC` results from the used RDC mechanism: Since a broadcast packet should reach all neighbors which are reachable, `ContikiMAC` has to re-broadcast the same packet for entire wake-up interval (120 ms) to make sure to reach all sleeping sensor nodes. During this period, the transmission medium is occupied, which results in a huge amount of collisions, as Figure 5.16 shows.

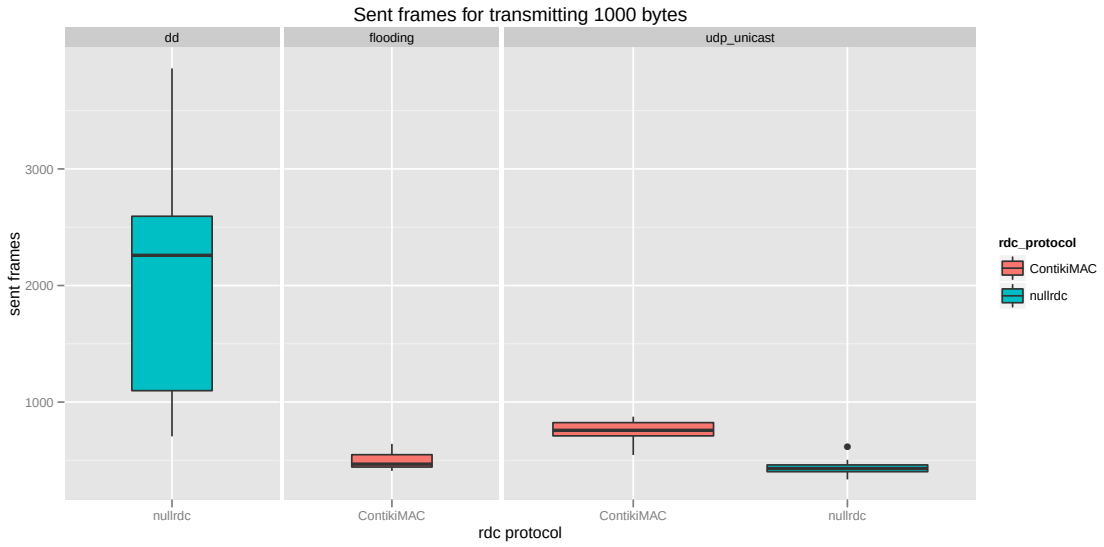

Figure 5.15: Sent frames for transmitting 1000 bytes in the *SmallNet* scenario

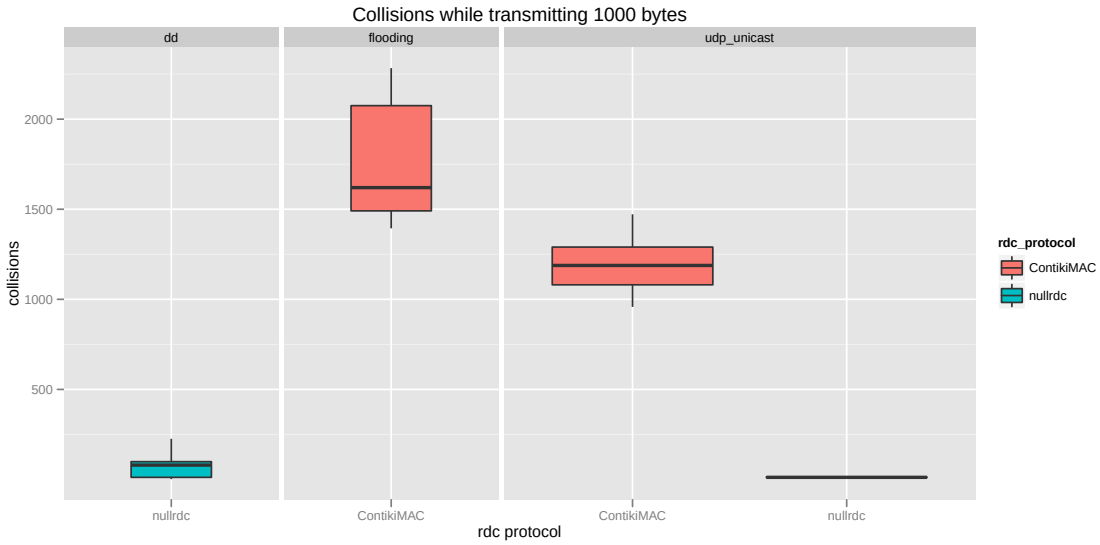

Figure 5.16: Collisions while transmitting 1000 bytes in the *SmallNet* scenario

5.5.4 Summary of Results

The results show strengths and weaknesses of the implemented communication protocols.

Flooding

Flooding is easy to implement and very reliable, but it is quite slow. If `NullMAC` is used, data transmission consumes a lot of energy. As shown in chapter 5.5.3, the number of sent frames and collisions is huge. Flooding wastes a lot of resources for the benefit of reliability and low protocol complexity.

MPR

MPR improves some drawbacks of Flooding by allowing only some nodes to forward data. This reduces collisions, thus leading to *faster* data transmission and *lower* energy consumption. The usage of the implemented packet queue with data aggregation of the data messages also helps reducing the number of duplicated sent packets. A packet queue is needed, first because as described in chapter 2.3.4, `NullMAC` does not cache any frames, so the packet buffering has to be implemented on the application layer. Secondly, the determined MPR set is not necessarily optimal. The flooding strategy implies the possibility of an intermediate sensor node to receive the same packet from multiple neighbors. Since the packet queue caches received packets, those duplicate packets would be re-broadcast needlessly. This is prevented by the aggregation of data messages by the used packet queue, which deletes those duplicate packets. MPR is more complex than Flooding. This is mostly the result of the *initialization phase* and the determination of the MPR set. With increasing number of adjacent sensor nodes, determination of the MPR set becomes costlier.

Directed Diffusion

Directed Diffusion is hard to implement, due to its protocol complexity. Especially the implementation of the gradient setup was time-consuming and difficult. Path discovery through broadcasting `interest_messages` does not work properly and is likely to return unreliable and unreasonably long paths. Due to these limitations, we experienced Directed Diffusion as very unstable. The advantage of Directed Diffusion is the independence of static and preconfigured routing tables. It is therefore able to distribute data in a priorly unknown topology.

UDP unicast

UDP unicast performs very well compared to the other protocols. Protocol complexity is very low and it is much faster, because it does not broadcast any packets. This leads to considerably less collisions and to a much faster data transmission.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The results show the effects of different caching strategies. Caching can definitely accelerate data transmission, but only if more than one data message is needed to transmit the payload. For a payload size of 70 bytes (fits into one data message), the costs for the caching does not compensate the reduction of the transmission time. Also, proactive caching does not prove itself an effective strategy, since it does not work without introduction of a *request limit*.

Besides the protocol logic, the used MAC protocol proves to be a major influence to the overall performance of a communication protocol. `ContikiMAC` significantly reduces energy consumption, but increases transmission time. `ContikiMAC` also has its weaknesses: It works only with a small number of directly reachable sensor nodes. The 40 sensor nodes of the *BigNet* scenario are definitely too many for `ContikiMAC` to handle. This limitation is caused by the RDC mechanism of `ContikiMAC`: Each node periodically wakes up for a short time, checking for the reception of a wake-up strobe. The time between two wake-ups is called t_{sleep}. As ContikiMAC has to synchronize all adjacent nodes in one time period t_{sleep}, there is a maximum number of nodes it can handle. Additionally, transmission timing has to be the more accurate, the more sensor nodes are in the perimeter. The second weakness of `ContikiMAC` is the way the Clear Channel Assessment (CCA) is done. Relying on the Received Signal Strength Indicator (RSSI) value makes `ContikiMAC` sensitive to signal noise. Therefore, intermediate sensor nodes tend to falsely assume a positive CCA (as the signal strength can easily exceed the threshold). A positive CCA causes the sensor node to sleep again without receiving any data, thus slowing down the data transmission. Beside these drawbacks, the results show `Contikimac` to be unsuitable for broadcast-based communication protocols, due to the necessity of resending the wakeup strobe for the entire sleep period to reach all adjacent nodes.

6.2 Future Work

As discussed in chapter 3.1, Flooding, MPR and Directed Diffusion implementation was done on the application layer. This simplified the implementation, but also increased the overhead. Also, µIP was chosen as communication stack, due to the required possibility of updating code from a

common desktop PC over the Internet. Instead of using µIP and implementing the protocols on the application layer, the RIME communication stack might be to the advantage, because of its simplicity.

Flooding, MPR and Directed Diffusion could be implemented on top of RIME, thus reducing the overhead of an application layer implementation. Additionally, using RIME could simplify the implementation, as it provides a lot functionality found in common network protocols, available as very thin layers. For example RIME supports the path discovery out of the box. Internet access could be enabled using a RIME proxy, translating IP packets to RIME packets and vice versa.

Bibliography

[1] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005.

[2] G. Wagenknecht, M. Anwander, and T. Braun, "MARWIS: A Management Platform for Heterogeneous Wireless Sensor Networks." *ERCIM News*, vol. 2009, no. 76, pp. 18–19, 2009.

[3] ——, "SNOMC: An overlay multicast protocol for Wireless Sensor Networks," in *Proc. 9th Annual Conf. Wireless On-demand Network Systems and Services (WONS)*, 2012, pp. 75–78.

[4] G. M. Dias, "Implementing a Reliable Overlay Multicast Protocol on Wireless Sensor Nodes," Master's thesis, Universität Bern, 2011.

[5] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed Diffusion for Wireless Sensor Networking," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 2–16, Feb. 2003.

[6] A. Qayyum, L. Viennot, and A. Laouiti, "Multipoint Relaying for Flooding Broadcast Messages in Mobile Wireless Networks," in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02) - Volume 9*, ser. HICSS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 298–.

[7] P. Hurni, M. Anwander, G. Wagenknecht, T. Staub, and T. Braun, "TARWIS — A Testbed Management Architecture for Wireless Sensor Network Testbeds," in *Proc. 7th Int Network and Service Management (CNSM) Conf*, 2011, pp. 1–4.

[8] "TmoteSky Datasheet." [Online]. Available: http://www.sentilla.com/files/pdf/eol/tmote-sky-datasheet.pdf

[9] "CC2420 Datasheet." [Online]. Available: http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=cc2420&fileType=pdf

[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04, vol. 0. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[11] A. Dunkels, “Full TCP/IP for 8-bit architectures,” in *Proceedings of the 1st international conference on Mobile systems, applications and services*, ser. MobiSys ’03. New York, NY, USA: ACM, 2003, pp. 85–98.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems,” in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys ’06. New York, NY, USA: ACM, 2006, pp. 29–42.

[13] A. Dunkels, “Rime - A Lightweight Layered Communication Stack for Sensor Networks,” in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, ser. Poster/Demo session, 2007.

[14] ——, “The ContikiMAC Radio Duty Cycling Protocol,” Swedish Institute of Computer Science, Tech. Rep. T2011:13, Dec. 2011.

[15] J. Polastre, J. Hill, and D. Culler, “Versatile Low Power Media Access for Wireless Sensor Networks,” in *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, USA, November 2004, pp. 95–107.

[16] M. Buettner, G. V. Yee, E. Anderson, and R. 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*, ser. SenSys ’06. New York, NY, USA: ACM, 2006, pp. 307–320.

[17] D. Moss and P. Levis, “Box-macs: Exploiting physical and link layer boundaries in low-power networking,” Stanford University, Tech. Rep., 2008.

[18] A. El-Hoiydi and J.-D. Decotignie, “Wisemac: an ultra low power mac protocol for the downlink of infrastructure wireless sensor networks,” in *Proceedings of the Ninth International Symposium on Computers and Communications 2004 Volume 2 (ISCC”04) - Volume 02*, ser. ISCC ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 244–251.

[19] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes, “Powertrace: Network-level Power Profiling for Low-power Wireless Networks,” Swedish Institute of Computer Science, Tech. Rep. T2011:05, Mar. 2011.

[20] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He, “Software-based On-line Energy Estimation for Sensor Nodes,” in *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, June 2007.

[21] “SQLite, a lightweight file-based database.” [Online]. Available: http://www.sqlite.org

[22] “GNU Plot, a portable command-line driven graphing utility.” [Online]. Available: http://www.gnuplot.info/

[23] “GNU R, a open-source statistical environment.” [Online]. Available: http://www.r-project.org/

[24] L. Sachs and J. Hedderich, *Applied Statistics*. Springer Gabler, 2012.