University of Zurich[UZH]

INTERNSHIP – Communication Systems Group, Prof. Dr. Burkhard Stiller

TinyIPFIX Aggregation in Contiki

Livio Sgier
Zurich, Switzerland
Student ID: 12-918-702

Supervisor: Dr. Corinna Schmitt
Date of Submission: May 9, 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

CSG
Department of Informatics
Communication Systems Group

Internship
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

TinyIPFIX is an application layer protocol to efficiently transmit data in a Wireless Sensor Network. The core idea is to decouple the header information from the actual payload in order to save valuable data transmission by sending them at different preset time intervals. This leads to a decrease of energy consumption compared to other application layer protocols. There exists an implementation of TinyIPFIX on the popular operating system Contiki. This work presents an extendible aggregation framework on top of TinyIPFIX with an implementation of two specific aggregation mechanisms: Message and data aggregation. Additionally, CoMaDa, an adaptive configuration, management and data handling framework, has been extended with Contiki support for manual device configuration and data visualization.

Zusammenfassung

TinyIPFIX ist ein Protokoll der Anwendungsschicht, welches zum Ziel hat Daten in einem Wireless Sensor Netzwerk effizient zu übermitteln. Die Grundidee besteht darin, den Header von den eigentlich zu übertragenden Nutzdaten zu entkoppeln. Die Header Informationen, sowie die Nutzdaten werden separat in einem festgelegten Zeitintervall gesendet. Die Umsetzung führt in einem typischen Wireless Sensor Netzwerk verglichen mit anderen Protokollen der Anwendungsschicht zu einem insgesamt reduzierten Datentransfer, was zu weniger Energieverbrauch führt. Es existiert eine Implementierung des TinyIPFIX Protokoll für das Betriebssystem Contiki. In dieser Arbeit wurde ein erweiterbares Framework für Aggregationsmechanismen entwickelt, welches auf TinyIPFIX beruht. Zwei spezifische Mechanismen, Nachrichten- und Datenaggregation, wurden implementiert. Zusätzlich wurde CoMaDa, ein Framework für Konfigurationen, Management und Datenaufbereitung für Wireless Sensor Netze, durch die Möglichkeit erweitert Sensoren für das Contiki Betriebssystem manuell über eine grafische Benutzeroberfläche zu konfigurieren und Daten durch verschiedene Visualisierungen grafisch darzustellen.

Acknowledgments

I would like to thank my supervisor, Dr. Corinna Schmitt, for her continuous assistance and inputs throughout this work. Furthermore, I would like to thank Dr. Corinna Schmitt and Prof. Burkhard Stiller, head of the Communication Systems Group at the University of Zurich, for making this internship possible.

Contents

Abstract **i**

Zusammenfassung **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Report Outline 2

2 Technologies, Context and Related Work **3**

2.1 Hardware Platforms 3

2.2 Contiki 4

2.3 Data Format TinyIPFIX 5

3 Design **9**

3.1 Components and Hardware Deployment 9

3.1.1 Border Router 11

3.1.2 Hardware Deployment 11

3.2 Routing Support 11

3.3 Aggregation 12

3.3.1 Message Aggregation 12

3.3.2 Data Aggregation 13

4 Implementation **15**

4.1 Routing Implementation 15

4.1.1 Simple UDP API 15

4.1.2 Service Registration 16

4.2 Modular Aggregator Software Design 17

4.3 CoMaDa Integration 18

5 Evaluation **21**

5.1 Memory Footprint 22

5.2 Performance 22

5.3 Energy Consumption 23

5.4 Limitations 24

6 Conclusion **25**

Chapter 1

Introduction

Wireless Sensor Networks (WSNs) [1] are a branch of the Internet of Things (IoT). Tiny devices, equipped with sensors registering and measuring their surroundings, are formed to a network, called WSN. They are used for environmental and energy sensing as well as for industrial monitoring, to name just a few of the vast number of application domains. They are often deployed in remote areas or away from fixed infrastructures and depend on mobile energy sources like batteries. The limited resources on wireless sensor nodes require specialized operating systems. Many of them exist, tailored specifically to support the scarce resources available. Two of the most widespread and adopted operating systems are TinyOS [10] and Contiki [11]. The resource-efficient data collection protocol TinyIPFIX [2] was an approach to save on transmission (and therefore energy) by decoupling the redundant meta information from the collected data, which is distinct for each measurement. TinyIPFIX has been implemented on the TinyOS and Contiki operating systems [3, 4] in a memory-saving manner to allow more applications and functionalities to be run on top of it. One desirable functionality is support for aggregation. Literature distinguishes two kinds of aggregation, namely message aggregation and data aggregation [8]. Message aggregation is carried out by specified nodes in the network, which combine several incoming messages into one message before forwarding it to the next hop in order to reduce overall traffic. Data aggregation, also called data pre-processing, apply a mathematical function on the input. Only the calculated output gets forwarded along the network. Possibilities for mathematical functions include the average, maximum or minimum value on all specified data, e.g., the average temperature of all surrounding sensors. Aggregation support has already been implemented in TinyOS [8].
CoMaDa is another component in this internship. It is a configuration, management and data handling framework for WSNs [22]. It allows for manual configuration of sensors and data visualizations of WSNs. *CoMaDa* is the gateway component in such networks, linking wireless and wired infrastructures together. Formally, it builds the bridge to the IoT.

1.1 Problem Statement

The design goal of TinyIPFIX is to reduce overall data transmission and energy consumption in a network by decoupling the header from the payload and transmitting them in different, pre-defined time intervals. The work done in [4] uses RPL [12] as an underlying routing protocol to transmit data from every node in the network towards the sink, e.g., the *Border Router*. Aggregation support provides further optimizations in this regard. Certain nodes in the WSN act as aggregators, which do not forward all incoming data right away but aggregate data from one or many sources before forwarding some aggregated form of the data towards the sink. The goal of this internship was to develop an aggregation framework on top of TinyIPFIX, which supports message and data aggregation. Also, manual configuration and data visualizations is implemented in the TinyOS operating system. It is a goal of this internship to integrate Contiki into CoMaDa such that the same features for TinyOS are available for Contiki.

1.2 Report Outline

The report is structured as follows: Section 2 broadly covers the building blocks used in this report. Section 3 is concerned with the design of the different software components as well as the aggregation framework, followed by implementation details of the software components, the aggregation framework and the *CoMaDa* integration in Section 4. Section 5 evaluates the framework with regard to different properties, such as memory footprint, performance and energy consumption. Existing limitations in the system are also mentioned. Finally, Section 6 presents a summary and draws a conclusion.

Chapter 2

Technologies, Context and Related Work

Several technologies were used in this project. This section introduces the most important ones ond puts them into perspective relative to each other. The physical layer is covered first. Then the operating system Contiki is broadly introduced before the relevant software and algorithms are investigated in more detail.

2.1 Hardware Platforms

Two different hardware platforms were used in this work, namely TelosB [5] by Advanticsys and OpenMote [6] by OpenMote Technologies. Both platforms can be seen in Figure 2.1. They vary considerably in their hardware capabilities. The following two subsections familiarize with the relevant technical specification of both sensor node platforms.

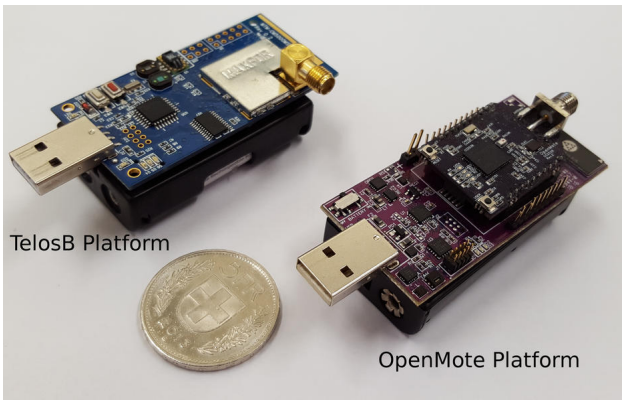

Figure 2.1: Hardware platforms and a 5 CHF coin for scale

The **TelosB** sensor node used in this project is the MTM-CM5000-MSP model. It is based on the original open-source TelosB platform design, developed and published by

the University of California, Berkeley. The microcontroller MSP430F1611 has a 16-bit RISC architecture and belongs to the Texas Instruments MSP430 family. It has 48 KB program flash memory and 10 KB RAM. The radio RF chip, Texas Instruments CC2420, is IEEE 802.15.4 compliant. The sensors on the TelosB sensor node are temperature, humidity and light. The sensor for measuring temperature and humidity is Sensirion SHT11 and the sensor for light (visible & infrared range) is Hamamatsu S1087 Series. The sensor node also has a 2xAA battery holder. [5]
Due to its scarce memory resources, the TelosB platform will be used for mere data collection and will not be equipped with aggregation logic. Chapter 5 looks at the memory footprint of the different software components in more detail.

The **OpenMote** hardware used in this project consists of the OpenMote-CC2538, which is plugged onto the OpenUSB. Both parts can be seen seperately in Figure 2.2. The core piece of the Openmote-CC2538 is the SoC (System on Chip) CC2538 from Texas Instruments. It has a 32-bit ARM Cortex-M3 microcontroller, which includes 512 KB program flash memory and 32 KB RAM. The OpenMote-CC2538 furthermore has a CC2520-like radio transceiver, which is fully compatible with the IEEE 802.15.4-2006 standard. The OpenUSB has the following sensors: Temperature/humidity sensor by Sensirion SHT21, a 3-axis accelerometer by Analog Devices' ADXL346 and a light sensor by Maxim Integrated's MAX44009 [6, 7]. As can be seen by the technical specification, the OpenMote platform has a larger program flash memory by a magnitude of more than ten and larger RAM by a magnitude of more than three. This makes the OpenMote platform suitable to be used as an aggregator, because the aggregation logic is more memory-intensive than the data collection and transmission software.

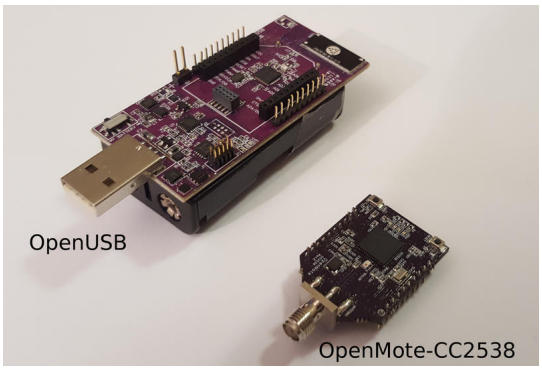

Figure 2.2: The OpenMote Platform

2.2 Contiki

Operating Systems for WSNs are different from traditional desktop or server operating systems like Linux or Windows and also different from powerful embedded systems like smartphones. The reason for this is the different requirements of WSNs, which are mainly limited hardware resources, low power, reactive concurrency and flexibility regarding the wide range of possible applications running on sensor nodes [14]. Among the many existent WSN operating systems, two of the most widespread ones are TinyOS [10] and

Contiki [11]. This work is based on Contiki whose important properties are shortly summarized:
Contiki is written in the C language and is based on an event-driven scheduler. The scheduler dispatches events to running processes. Dispatched events run to completion and cannot be interrupted by the scheduler. The problem with purely event-driven programming models is the lack of blocking waits. This necessitates the use of state machines to describe control flow. Programs written in state machines are difficult to write, maintain and debug. Adam Dunkels came up with a programming abstraction that allows for blocking waits and thread-like programming while at the same time minimizes memory overhead for each thread as much as possible. This reduces considerable complexity to write programs. The abstraction is called Protothread [15]. It allows for stack-less threads with minimal memory overhead of 2 Bytes per Protothread. Since Protothreads are stack-less, preemption is not possible. To still allow preemptive multi-threading, programs can be optionally linked with a library that sits on top of the event-driven kernel and supports preemption. Linking the library should be used carefully because of the additional memory overhead.

2.3 Data Format TinyIPFIX

TinyIPFIX is an application-layer protocol designed for resource-constrained wireless sensor nodes to transmit data in a WSN. The name TinyIPFIX comes from the IPFIX protocol, which stands for Internet Protocol Flow Information Export [20]. IPFIX is used to transmit traffic flow information to facilitate services such as measurement, accounting and billing in IP networks. It does so by splitting information into template and data records. Information flows unidirectionally from exporting nodes towards collecting nodes. TinyIPFIX adapted the idea of splitting information into meta data and actual payload and also incorporated the unidirectional, push-based flow of data within the network. Measurements [3] have shown that the transmission efficiency of TinyIPFIX compared to other approaches like Type-Length-Value [16] is promising.

A template record in the TinyIPFIX protocol uniquely describes the specific model of a sensor by an indivdual Information Element Identifier and an Enterprise Number, which are registered with the Internet Assigned Number Authority (IANA). This means the template record represents the meta information for a sensor reading. This ensures interoperability and adaptability to other platforms. A template record always references a unique data record, which matches the sensor. Whether the Enterprise Number is set is

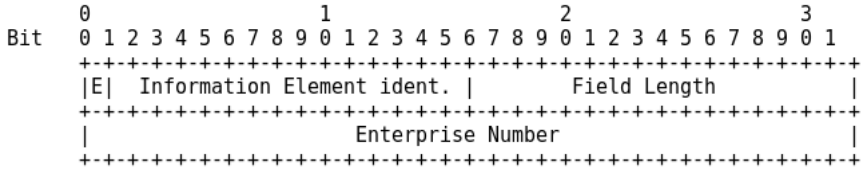

Figure 2.3: Template Record [2]

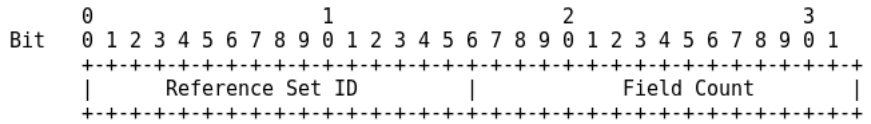

Figure 2.4: Template Set Header [2]

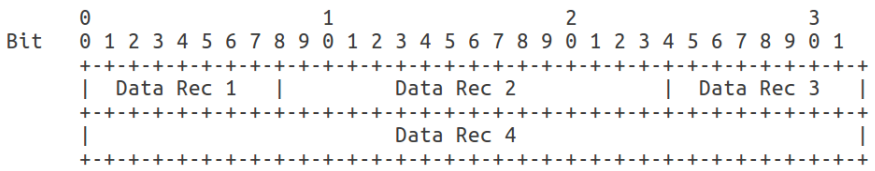

Figure 2.5: Data Set consisting of multiple Data Records [4]

optional and can be distinguished with the value of the first bit of the Information Element Identifier field. The Field Length field denotes the size of the sensor data in octets. Figure 2.3 shows the structure of a template record. Several template records make up a template set. On top of the resulting template set sits the template set header (Figure 2.4), which denotes the data set ID it is referencing as well as the field count, which is the total number of records in the set.

A data record only consists of the data itself. By knowing the 1-to-1 mapping between a template set and a data set no further information is necessary to uniquely attribute the data to the corresponding sensor. The length of each data record is a multiple of one byte. Several data records combined result in a data set. An example set with variable sized data records can be seen in Figure 2.5. Data sets do not have a data set header because a receiver has to receive a template set first, which references this data set to know how to interpret the data. A message header sits on every TinyIPFIX message. This header looks very different from the IPFIX header, but as stated before, IPFIX and TinyIPFIX have other tasks to fulfill and, therefore, underlie different design desicions. Figure 2.6 shows the TinyIPFIX message header with all corresponding fields.

The first two bits are two flags. If the first bit, denoted as A, is set, then the optional Ext. SetID (Extended Set ID) field is present in the header. The same logic applies to the flag B and the optional Ext. Sequence Number field. The SetID lookup field is

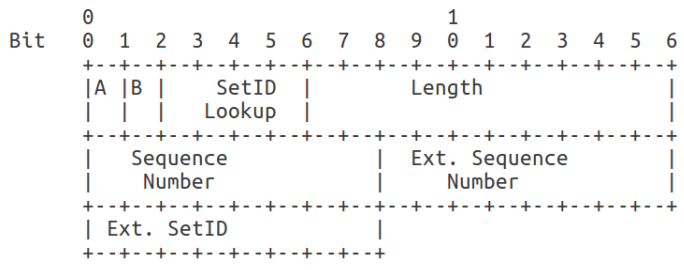

Figure 2.6: TinyIPFIX Message Header [2]

always present and gives information about what kind of information is being sent with the actual TinyIPFIX message. The standard uses the following shortcuts: Value = 1 means SetID = 2 and the message contains a template definition. Value = 2 means SetID = 256 and the message contains a data set for Template 256. Value = 0 and Value = 15 are for special use and Value = 3-14 are reserved for future use.

The following chapter is concerned with the design decisions for the implementation of the aggregation framework under Contiki for the above-mentioned hardware platforms.

Chapter 3

Design

The aggregation framework is based on an existing TinyIPFIX implementation [4] for Contiki and is inspired by the aggregation implementation in TinyOS [8]. This chapter focuses on a high-level overview of the design decisions taken in this internship. Section 3.1 introduces the different software components and their deployment on hardware. Section 3.2 targets routing, specifically which data path TinyIPFIX messages take from collector to sink. Section 3.3 is concerned with the design of message and data aggregation on top of TinyIPFIX.

3.1 Components and Hardware Deployment

There are three distinct software components present, namely *Collector*, *Aggregator* and *Border Router*. The functionality of each of them is described in more detail.

A *Collector* on Contiki has been implemented in [4]. Its task is to periodically transmit its pre-defined template set and updated measurements from its sensors towards the *Border Router*. The *Collector* code has been extended in order to allow it to communicate with specific *Aggregator* nodes via unicast.

Figure 3.1 shows the architecture of the software. As the node becomes alive, either by plugging 2xAA batteries into the battery holder or plugging it into a powered USB slot, the controller module initiates the networking module to set up a connection to the *Border Router* and initalizes the TinyIPFIX system. The platform is known at compile-time. The platform is transparent in the way that TinyIPFIX itself is not aware of the platform being used. If TinyIPFIX requests a template set or a data set, then the request is directly handled by the platform code. The advantage of this design is that the TinyIPFIX code never has to be modified and that only a few lines of code have to be written for a new platform to be successfully integrated. In this example the TelosB platform is active, but by recompiling with different command line options the OpenMote or other, not yet implemented, platforms can be used.

The *Aggregator* receives incoming messages from the *Collectors* and stores them in RAM. It then aggregates the received measurements before sending them along.

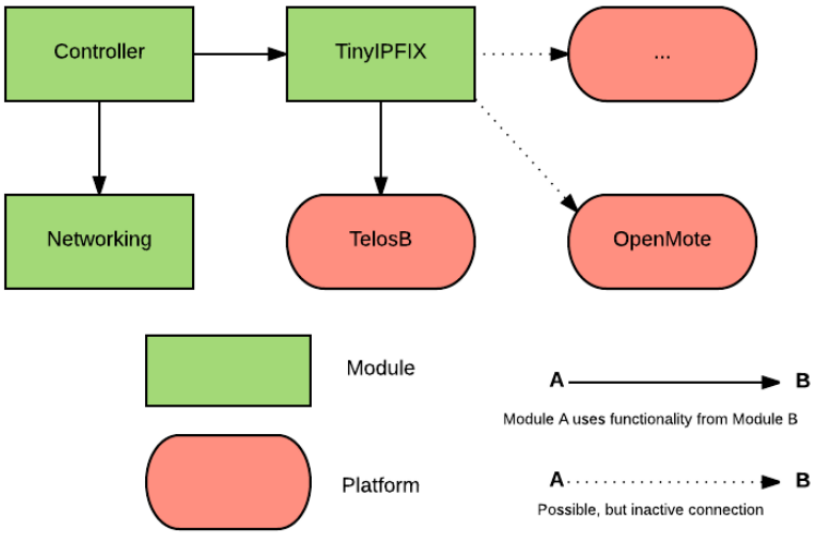

Figure 3.1: *Collector* Software Design

Figure 3.2 shows the components of the *Aggregator* software. As soon as the *Aggregator* comes to life, the controller module tasks the networking module to initialize two UDP connections: one incoming connection to receive messages from the *Collectors* and one outgoing connection to forward the aggregated messages towards the *Border Router*. The aggregation module, which will be further explained in the next subsection, is responsible for calculating the aggregated template set, which it periodically forwards to the networking module by which it gets sent off to the *Border Router*. The networking module forwards all incoming messages to the aggregation module to apply aggregation on the data sets. Thus, this communication is bidirectional. Only the data sets received from the *Collectors* are acted upon, the template sets are discarded. Finally, the aggregation module needs access to TinyIPFIX specific tasks such as message header creation.

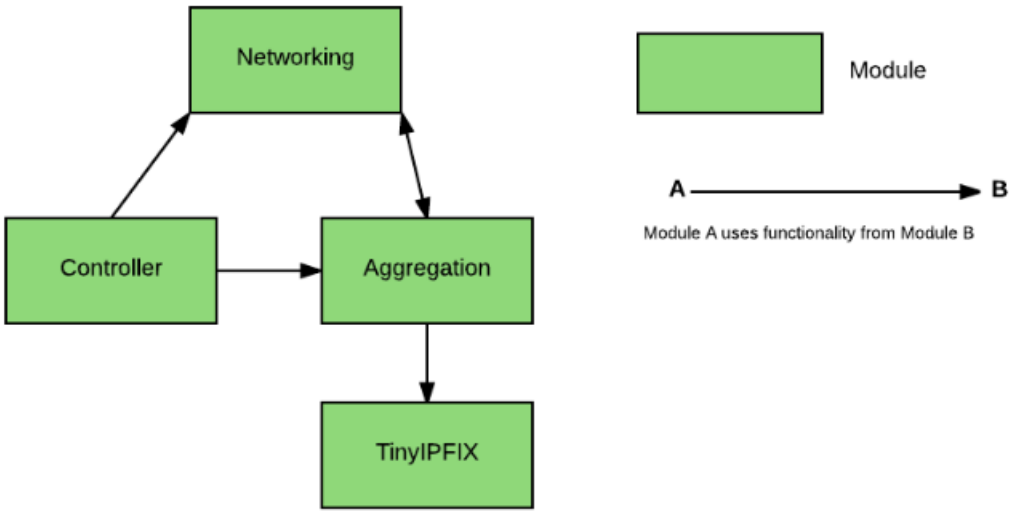

Figure 3.2: *Aggregator* Software Design

3.1.1 Border Router

The *Border Router* is an application developed by Contiki developers [19]. It receives template sets and aggregated messages from *Aggregators* only. It is programmed in a way that it listens for all incoming messages, not only messages specifically sent to its IP. This is resource-consuming, but the *Border Router* is connected to a computer by default and, thus, does not depend on batteries or other mobile energy sources. The fact that the *Border Router* receives messages not only destined to its IP allows the *Aggregator* to forward its outgoing messages to a default IP within the network. However, there are downsides to this approach, which will be discussed in more detail in the evaluation chapter.

3.1.2 Hardware Deployment

Due to the differences in memory footprint and application logic, the simpler *Collector* component is deployed on the TelosB platform. As mentioned in [4], TelosB only supports Contiki versions up to 2.7 because of its constrained ROM space. OpenMote on the other hand supports more up-to-date and much improved versions [21] of the operating system. Fortunately, the whole stack is backwards-compatible to version 2.7, which makes a heterogeneous deployment consisting of TelosB and OpenMote hardware possible. The *Aggregator* and *Border Router* are both deployed on OpenMote hardware. The increased ROM space of the OpenMote platform allows for larger aggregated messages.

3.2 Routing Support

The implemented routing mechanism for this aggregation framework is static. This decision is based on the assumption that the nodes are at fixed positions throughout the deployment. Furthermore, the most likely scenarios are measurements based on locality where a fixed mapping between *Collectors* and *Aggregators* is desirable, e.g., all sensors within a room communicate with an *Aggregator* responsible for that room. Even though dynamic routing adds flexibility, a static routing approach reduces overhead in both the *Collector* and the *Aggregator* compared to a dynamic approach. Figure 3.3 shows an architectural overview of the system with the components described in the previous section and the static routing in place. The *Collectors* communicate unidirectionally with the *Aggregator*, i.e., the *Collectors* are not listening to incoming messages. Instead of hardcoding the IP addresses of the *Aggregator* directly into the source code, the *Collectors* as well as their corresponding *Aggregator* subscribe to a *service* which groups them logically into a subnetwork. The technical details follow in Section 4.2. The service registration concept allows a dynamic replacement of the *Aggregators* without reprogramming the *Collectors*. The *Aggregators* listen on incoming messages from *Collectors* and send the aggregated messages to the *Border Router*, depending on the mode of aggregation as well as pre-defined configuration constants set at compile-time. The single *Border Router* receives messages from the aggregation nodes only and does not send messages back into

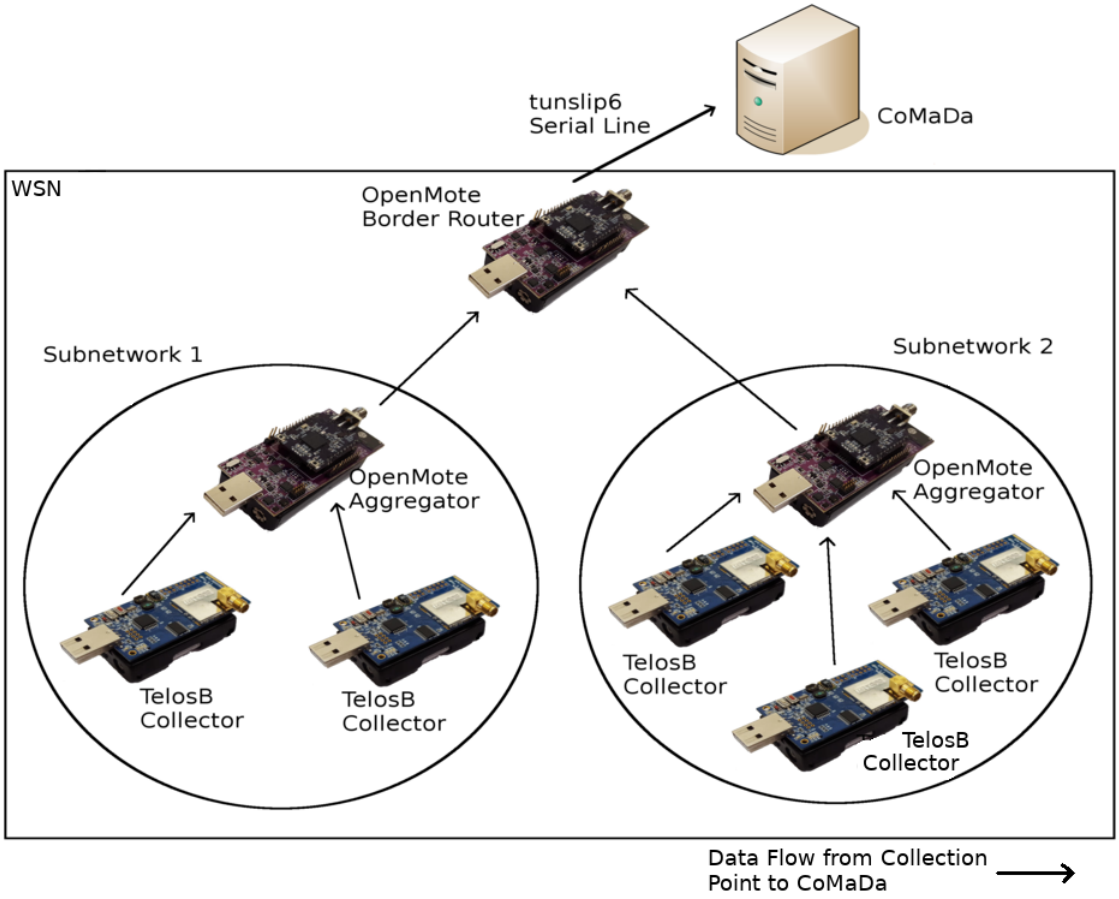

Figure 3.3: Architecture of the Deployed Aggregation Framework

the WSN. The *tunslip6* application [17] is part of the Contiki operating system and allows to bridge IP traffic between two networks. *tunslip6* creates a virtual network interface on the host side, e.g., the computer the application runs on, and uses the Serial Line Internet Protocol (SLIP) [18] to encapsulate and pass traffic from the network interface of the *Border Router* to the virtual network interface on the host side. This is needed to feed the data into CoMaDa [22] for further manipulation.

3.3 Aggregation

This section explains how to build the above-mentioned aggregation mechanisms on top of TinyIPFIX. The following mechanisms are run on the *Aggregator* node. Technical details are left for the implementation chapter.

3.3.1 Message Aggregation

The goal of message aggregation is to reduce traffic by combining separate elements. Message aggregation, in contrast to data aggregation, does not alter or manipulate the

data but only combines them into a larger message. As described in Section 2.3, every TinyIPFIX payload needs to have a corresponding template definition which needs to be sent additionally to the payload in a separate message such that the receiver of the data can correctly parse the payload. The *Aggregator* node is responsible for constructing the aggregation template. Since message aggregation is just the concatenation of individual messages, the resulting template is just the concatenation of several times the individual template set received from the *Collector*. The amount of concatenations corresponds to the *degree of aggregation (doa)*.
To illustrate the mechanism, consider Figure 3.4: *a)* shows a template set with three template records. This is the template that is sent from the *Collector* towards the *Aggregator*. *b)* shows an aggregated template set with a degree of aggregation equal to two, which is periodically broadcast from the *Aggregator* to the *Border Router*. As mentioned in Section 2.3, a template set header as well as a message header is placed on top of each TinyIPFIX message. By aggregating individual messages, transmission density can be reduced by including the header once for the aggregated template. As for the payload, the *Aggregator* waits for the amount of data messages as specified in the *degree of aggregation* parameter from the *Collectors* within its subnetwork, concatenates them, calculates the corresponding message header before sending the aggregated data message towards the *Border Router*.

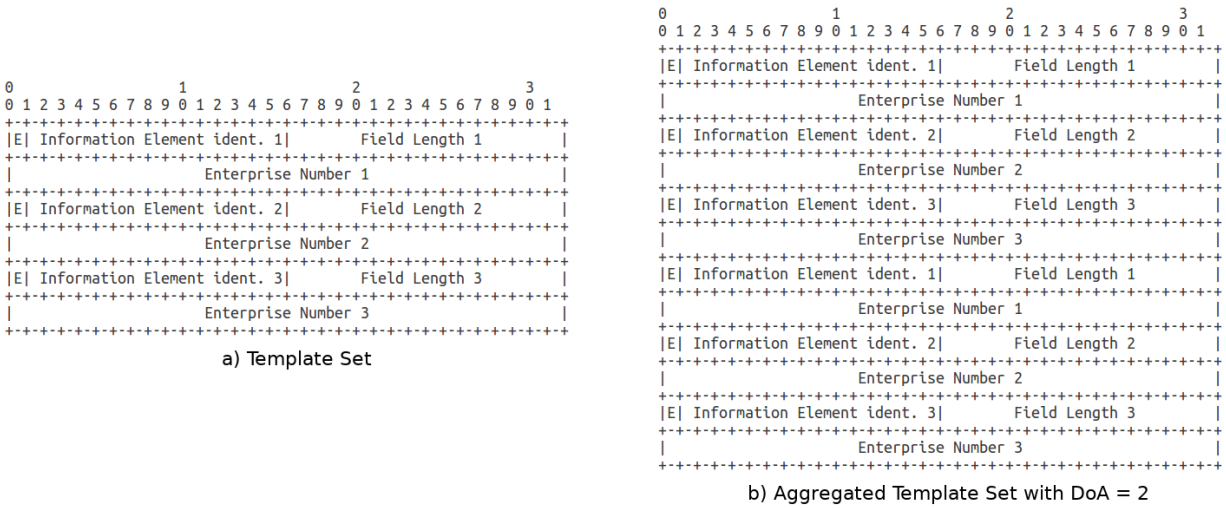

Figure 3.4: Message Aggregation

3.3.2 Data Aggregation

Data aggregation focuses on a subset of the data. In this internship one type of measurement is considered, but this can easily be extended to multiple measurements or the whole template set itself. The following parameters are needed and set at compile-time for data aggregation:

- **Measurement Type**: This value corresponds to the type of measurement or sensor

that is aggregated. Examples include temperature, humidty or battery level, among others.

- **Degree of Aggregation**: This is similar to message aggregation. The *degree of aggregation* is the amount of input messages the aggregation function operates on.

- **Aggregation Function**: This is a mathematical function which takes as input a number of data messages, which is specified in the *degree of aggregation* parameter, and outputs a single value. Examples of aggregation functions include *average*, *min* or *max*, among others.
  In Figure 3.5, *a)* refers to the template set received from a *Collector* and *b)* shows the aggregated template set. This template set is periodically sent to the *Border Router*. The measurement type in this example is *element 2*. Once the *Aggregator* receives the amount of data sets specified in the *degree of aggregation* parameter, it extracts the relevant data, aggregates them according to the specified *aggregation function* before calculating the message header and seding the aggregated message towards the *Border Router*.

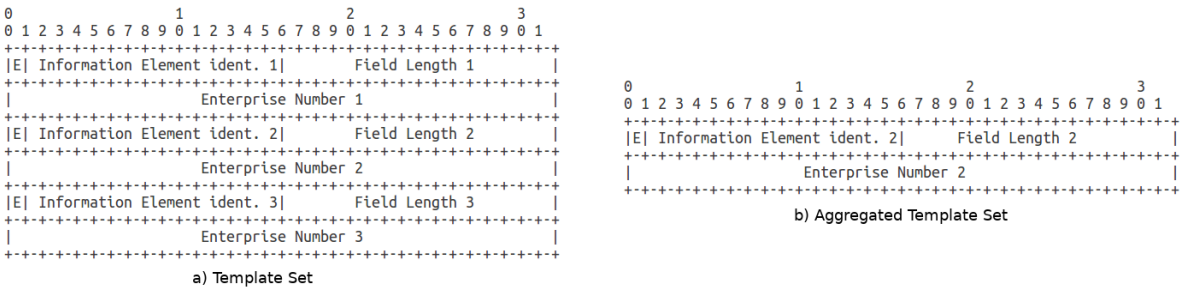

Figure 3.5: Data Aggregation

Chapter 4

Implementation

This chapter is concerned with implementation details, ranging from Contiki libraries and services, to the implementation of the aggregation framework to implementation details concerning the CoMaDa integration.

4.1 Routing Implementation

This section is concerned with the technical implementation of the routing mechanism. The building blocks, the *Simple UDP API* on the network layer and the *Service Registry App* on the application layer are provided by the Contiki codebase.

4.1.1 Simple UDP API

Contiki provides a simplified UDP API, called *Simple UDP API*, which simplifies networking tasks for the programmer. The API is shortly introduced in the following paragraphs.

Interfacing with the *Contiki UDP API* requires a programmer to handle socket creation, binding and listening on the receiving end. Additionally, a server has to listen for a *tcpip_event* for all UDP connections and can distinguish between different UDP connections only after the event has occured. The *Simple UDP* module simplifies this approach by using different user-defined callback functions for each distinct connection.

Listing 4.1: Registering a UDP connection

```
1 int simple_udp_register(struct simple_udp_connection *c,
2                         uint16_t local_port,
3                         uip_ipaddr_t *remote_addr,
4                         uint16_t remote_port,
5                         simple_udp_callback receive_callback);
```

Listing 4.1 shows the definition of the *simple_udp_register* function. The *struct*, called *simple_udp_connection*, has attributes of all necessary information about the connection, such as the callback function, the local port for a receiver and additionally the remote port and IP address of a receiver for a sender. The functions essentially writes or references the other arguments to the *simple_udp_connection*, which is then used as a handle for further interfacing with the *Simple UDP API*. The *simple_udp_callback* is a function pointer. This function is called as soon as an incoming packet for this connection is received. The return value indicates whether the UDP connection was successfully allocated.
Listing 4.2 shows how to use the above created UDP connection to send data. The first argument is the *struct* pointer for a previously created UDP connection, while the second and third argument are the data to be sent and its length, respectively. The return value indicates whether the sending operation was successful. A slight variation of the above function (*simple_udp_sendto*) takes the address of the receiver as an additional argument in case this information was not known during registering of the UDP connection. This modified function, the UDP registering function and the specific callbacks is all that is needed for all of the communication in this work.

Listing 4.2: Sending data over the UDP connection

```
1 int simple_udp_send(struct simple_udp_connection *c,
2                      const void *data, uint16_t datalen);
```

4.1.2 Service Registration

This subsection covers the *Service Registration* application that allows to logically group a subnetwork, as shown in Figure 3.3. It is not part of the Contiki operating system stack but an additional application included in the codebase, which can be found in the folder *apps/servreg-hack*. This application provides means for registering, disseminating, and looking up services. Every node, except the *Border Router*, in the WSN runs this application as a seperate thread in the background. An *Aggregator* registers a service with the function call displayed in Listing 4.3.

servreg_hack_id_t is a *typedef* for a 8-bit integer and the *service_id* is a global identifier. The IP address indicates which node the service belongs to, in this case the IP address of the *Aggregator* itself. All *Collectors* initialize the application by calling the *servreg_hack_init* function. Before the *Collector* transmits data towards the *Aggregator*, it fetches the address responsible for the service (Line 1), e.g., the IP address of the *Aggregator* by supplying the global 8-bit identifier *SERVICE_ID*, and sends the data off. If a service has been registered, the address is queried and the data is sent to that address, as shown in Line 3 of Listing 4.4. The function *simple_udp_sendto* has been described in the previous subsection.

Listing 4.3: Sending data over the UDP connection

```
1 void servreg_hack_register(servreg_hack_id_t service_id, const uip_ipaddr_t
    *addr);
```

Listing 4.4: Sending data over the UDP connection

```
uip_ipaddr_t addr = servreg_hack_lookup(SERVICE_ID);
if(addr != NULL)
   simple_udp_sendto(...);
```

The renewed querying before every transmission adds overhead but simplifies the scenario in which *Aggregators* get replaced. This way *Collectors* do not need to be reprogrammed and redeployed.

4.2 Modular Aggregator Software Design

The aggregation framework has been designed in a modular fashion. The goal of using a modular design is easier maintenance and the possibility to further extend the system with additional functionality. The latter was the motivation to design the aggregation module in a modular fashion, as can be seen in Figure 4.1. Message aggregation and data aggregation are specific implementations of the aggregation interface. The aggregation interface specifies two functions which have to be implemented by every aggregation mechanism. These two functions can be seen in Listing 4.5. The function *get_aggr_template* returns a pointer to memory where the aggregated template set is stored. Because the template set does not change for the duration of the deployment, no data has to be transferred to the aggregation module for further calculation or manipulation. On the other hand, data sets are further manipulated in the aggregation implementation by passing them to the selected implementation using the *calc_aggr_payload* function. This can be either the discussed mechanisms from Section 3.3 or new ones not further described here. The specific implementation can be selected by setting a compiler variable in the *Aggregator Makefile* accordingly.

Listing 4.5: Aggregation Implementations adhering to the Aggregation Interface

```
uint8_t *get_aggr_template();
void calc_aggr_payload(uint8_t* data, uint16_t datalen);
```

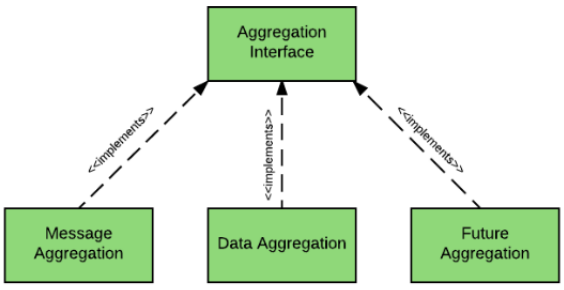

Figure 4.1: Aggregation API

4.3 CoMaDa Integration

A further goal of this internship was to integrate the developed aggregation framework into CoMada v1.1 for Contiki. This consists of two parts, namely manual configuration of the nodes as well as forms of live visualization for a deployed WSN.

Instead of compiling and deploying software for the nodes over the command line interface, a graphical user interface (GUI) is more comfortable when having to specify several parameters. Figure 4.2 shows four buttons, each of which opens a new panel, letting the user interact with the physically connected wireless nodes directly. *Collector*, *Aggregator* and *Border Router* buttons let the user specify parameters and compile and deploy the software components onto the connected hardware with one click. The *Tunnel* button lets a user start a virtual network interface to bridge the incoming traffic of the *Border Router* and feed it into CoMaDa for further processing. To illustrate the process of specifying deployment parameters, Figure 4.3 displays the panel when the *Aggregator* button is clicked and the *Data Aggregation* radio button selected. As stated in Section 3.3.2 three parameters can be specified which lets the user choose the *Degree of Aggregation*, *Aggregation Function* and *Aggregated Sensor*, the specific sensor which is to be aggregated. The specification of the subnetwork corresponds to Figure 3.3. The *Collector* panel has an identical input field. The source code location points to the local path on which the source for the specific software component resides. After specifying all parameters, the *Deploy* button is clicked, which compiles and deploys the software on the connected node.

New CoMaDa modules were developed to add live visualization for Contiki. The CoMaDa application is fairly modular, such that the process of adding new modules and drivers is straightforward. The *WSNFramework* class is the entry point of the application and lets the programmer choose which modules and drivers should be active before starting the application. As soon as the IP tunnel is started and the virtual network interface created, a live packet feed can be found under the *Visualization* → *Data-Packets* tab.
Figure 4.4 shows two TinyIPFIX messages: The top message is the template set and the bottom message is the aggregated message using message aggregation with an aggregation degree of 2. The node ID of the *Aggregator* is *38891* and can be seen in the first line of the aggregated message. The two *Collectors* with the node IDs *29066* and *22322* belong to the same subnetwork as the *Aggregator*. Figure 4.5 shows the template set for data aggregation, followed by three consecutive aggregated messages. The parameters for the data aggregation were set to *Degree of Aggregation = 2*, *Aggregation Function = Average* and *Aggregated Sensor = Temperature* (cf. Figure 4.3).

Figure 4.2: CoMaDa Configuration Panel

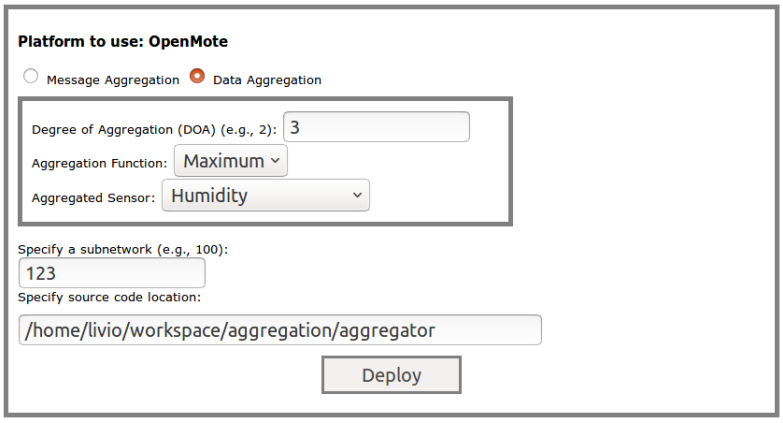

Figure 4.3: CoMaDa Configuration Data Aggregation Panel

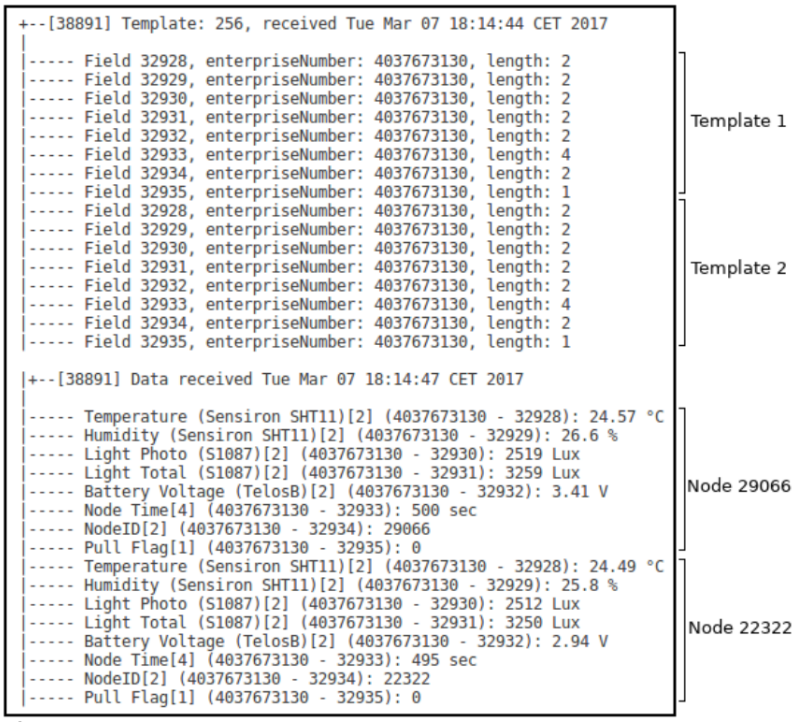

Figure 4.4: Message Aggregation Live Visualization in CoMaDa

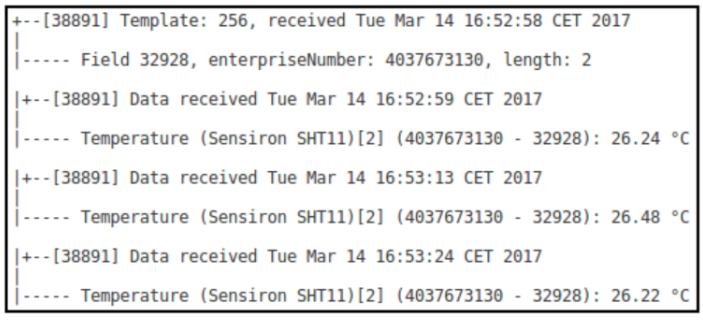

Figure 4.5: Data Aggregation Live Visualization in CoMaDa

Chapter 5

Evaluation

This chapter measures several qualitative properties, such as memory footprint, energy consumption and performance. Section 5.1 lists the measurement of the memory footprints of all components and compares the numbers with other work. Section 5.2 is concerned with performance of aggregated messages versus non-aggregated messages in terms of transmission quantity. Section 5.3 looks at energy consumption. The chapter concludes with limitations in the aggregation framework in Section 5.4.

Figure 5.1 shows a floor plan of the 2nd floor of the building at Binzmühlestrasse 14, 8050 Zurich. A heterogeneous WSN was deployed with all components listed in Section 3.1. The deployment consists of one *Border Router*, two *Aggregators*, and four *Collectors*. The numbers written next to each *Aggregator* and *Collector* correspond to the subnetwork they are placed in, as shown in Figure 3.3. *Collectors* belonging to the same subnetwork take measurements in the same room, before forwarding the data via unicast connection to their corresponding *Aggregator*. A measurement was run for 37 minutes with a transmission success rate of 100%.

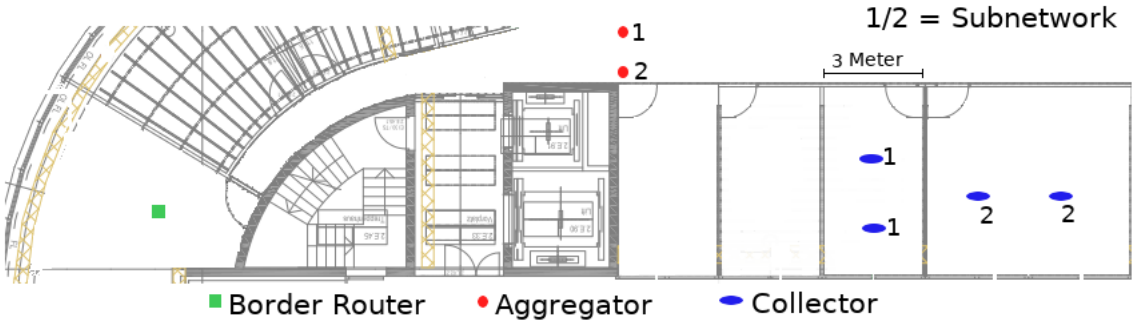

Figure 5.1: Deployment Scenario

5.1 Memory Footprint

Since wireless sensor nodes are highly constrained in their memory, special care has to be put into optimized programming. As mentioned in Section 2.1, the hardware platforms used in this work differ greatly in their memory resources. The justification of which software component is placed on which hardware platform was given in Section 3.1: The *Collector* is deployed on TelosB hardware, and the *Aggregator* as well as the *Border Router* are placed on the OpenMote hardware. The code of the *Collector* is a modified version of the implementation in [4]. The comparison of the memory footprint of the two implementations can be seen in Table 5.1. The bottom row is the *Collector* implementation of this internship, while the middle row is the memory footprint from the implementation in [4]. The numbers are slightly different for ROM and RAM memory. This is because of some code restructuring done in this implementation as well as introduced overhead from the *Service Registration* application.

	RAM [Byte]	ROM [Byte]
TelosB Maximum Memory [5]	10'000	48'000
TinyIPFIX *Collector* from [4]	6'492	41'326
TinyIPFIX *Collector* (cf. Section 3.1.1)	6'412	43'202

Table 5.1: Memory Footprint Comparison of Two TinyIPFIX *Collectors* on TelosB

An OpenMote node has much larger ROM and RAM sizes. Table 5.2 shows the physical amount of memory available in the first row and the two software components and their memory consumption in the 2nd and 3rd row. It does not make much sense to compare Table 5.1 and Table 5.2 because TelosB and OpenMote are based on different underlying hardware (cf. Section 2.1). However, fitting the *Aggregator* logic onto the TelosB might have been difficult. In contrast, aggressive optimizing of the code for the OpenMote platform is not necessary because the code uses just a small amount of the overall available memory.

	RAM [Byte]	ROM [Byte]
OpenMote Maximum Memory [6]	32'000	512'000
Aggregator (cf. Section 3.3)	17'217	47'603
Border Router [19]	18'672	49'440

Table 5.2: Memory Footprint of the *Aggregator* and *Border Router* [19] on OpenMote

5.2 Performance

In this section we consider the savings on transmission if aggregation is used. Specifically, we look at the communication path between the *Aggregator* and the *Border Router* and compare the savings relatively to a mere forwarder of all incoming data, as seen in [4].

We look at data and message aggregation separately. The underlying design of both aggregation methods was elaborated in Section 3.3.
In the case of **message aggreagtion**, for templates as well as measurement data, the following amount of Bytes are saved for *DOA* number of messages (*DOA = degree of aggregation, MSG_HEADER_SIZE = TinyIPFIX message header size in Bytes*):

$$(DOA - 1) \times MSG_HEADER_SIZE \tag{5.1}$$

Assuming the following pre-set constants: *DOA* = 3, *MSG_HEADER_SIZE* = 3 Bytes, *Data Set* = 17 Bytes, *Template Set* = 64 Bytes, a total amount of 10% can be saved in case of a data set and 3% for template sets.

The numbers look a lot different for **data aggregation**. An additional parameter, *SIZE_AGGR_DATA*, describes the size of the aggregated data: For data sets, this corresponds to the size of the measurements to aggregate and for template sets, it corresponds to the template records to which the data to aggregate belongs to. The following number of Bytes are saved for *DOA* number of messages (*PL = Payload*):

$$(DOA - 1) \times (MSG_HEADER_SIZE + PL) + (PL - SIZE_AGGR_DATA) \tag{5.2}$$

Considering the following pre-set constants: *DOA* = 4, *MSG_HEADER_SIZE* = 3 Bytes, *Data Set* = 17 Bytes, *Template Set* = 64 Bytes, *SIZE_AGGR_DATA* = 2 Bytes for data sets and 8 Bytes for template sets, a total amount of 93.75% transmission is saved for data sets and 95.9% is saved for template sets.

Considering the fact that the energy cost for transmitting one bit is more than processing 100 instructions on a Berkeley sensor node [23], transmission savings is very important for energy preservation, which makes the aggregation approach important for large-scale deployments.

We just considered the transmission savings for the TinyIPFIX application layer. However, using the aggregation framework presented not only leads to less data sent per packet, but also less packets travelling over the network in total. Specifically, only one packet in *DOA* number of received packets gets sent from the *Aggregator* to the *Border Router*. The overhead of each packet sent varies considerably, depending on the underlying protocols used. Using a traditional IPv6/UDP header adds an overhead of 48 Bytes. Therefore, 6LoWPAN [24] and other resource-saving protocols might be worth considering.

5.3 Energy Consumption

The energy measurement setup can be seen in Figure 5.2. A 10 Ohm resistor in combination with an applied voltage of 2V was used to measure the current flow. For the *Collector* running on the TelosB platform, the energy consumption in the idle state is 35.4 mA. The same amount of current was drawn in the event of sending template and data sets, namely 36.8 mA. For the *Aggregator* running on the OpenMote platform, the idle energy consumption is 39.2 mA and sending data, independent of template or aggregated data sets, resulted in a current flow of 39.7 mA. Surprisingly, the *Degree of Aggregation* had little effect on the energy consumption.

Figure 5.2: Energy Measurement Setup [4]

5.4 Limitations

A requirement for this internship was to use the available *RPL Border Router* [19] as the sink. This introduced limitations in the actual deployment settings. The problem is the following: Whenever the *Collectors* were within sending range to the *Border Router*, the message transmitted were directly received by the *Border Router*, without taking the necessary hop over the *Aggregator*. This happened even though the message was sent to the specific IP address of the *Aggregator*. This is because the *Border Router* is constantly sensing the air for all messages within the network, regardless whether they are targetted at the *Border Router* specifically. Additionally, there must be redundancy prevention mechanisms built into RPL that prevent the message being received at the *Border Router* and the *Aggregator*. However, this could not be shown. The result of this limitations was that the *Collectors* needed to be placed at a distance such that the sending data would not be receieved by the *Border Router* directly, but had to hop over the *Aggregator*. Moving the *Collectors* closer towards the *Border Router* in the deployment scenario shown in Figure 5.1 resulted in a direct communication line between the *Border Router* and the *Collectors*.

Chapter 6

Conclusion

The scope of this work was two-fold: Firstly, to develop a lightweight and energy-saving aggregation framework for Contiki which is at the same time applicable to heterogeneous WSNs. Secondly, a CoMaDa integration of the aggregation framework to facilitate manual configuration and live visualization.

After the motivation for this project and a short introduction to the different technologies that came together in this project, several design decisions were presented and argued for. These consisted of the design aspects of which hardware platforms can and should be used for which software components. A high-level concept of the intra-WSN routing was explained and the concept of aggregation in general and message and data aggregation specifically on top of TinyIPFIX was elaborated. The subsequent implementation chapter incorporated these design decisions into code and focused on the technical details, such as specific Contiki APIs and libraries offered by Contiki that were leveraged in this work. Also, a modular concept for the aggregation framework was presented with explanations on how to easily incorporate custom aggregation techniques. The CoMaDa integration combined all moving parts and explained how the different software components can be deployed on the hardware platforms with parameters that are entered over a GUI. The live visualization showed a live deployment for the different aggregation strategies.

The evaluation chapter considered the memory footprint in comparison to earlier work, the performance of all relevant software components, the energy consumption as well as the limitations that were given by the assumptions of this internship.

One can think of future work building up on this aggregation framework. One example would be to develop a security architecture built on top of TinyIPFIX that leverages the presented aggregation framework.

Bibliography

[1] H. Karl, A. Willig: *Protocols and Architectures for Wireless Sensor Networks*, John Wiley & Sons, ISBN: 978-0-470-09510-2, April 2005.

[2] C. Schmitt, B. Stiller, B. Trammell: *TinyIPFIX for smart meters in constrained networks, Internet Draft*, https://tools.ietf.org/html/draft-schmitt-ipfix-tiny-00, June 2016.

[3] T. Kothmayr: *Data collection in Wireless Sensor Networks for Autonomic Home Networking*, http://kothmayr.net/wp-content/papercite-data/pdf/kothmayr2010bachelor.pdf, 2010.

[4] L. Sgier: *Optimization of TinyIPFIX Implementation in Contiki and Realtime Visualization of Data*, Software Project, Communication Systems Group, Department of Informatics, University of Zurich, Zurich, Switzerland, October 2016.

[5] Advanticsys: *TelosB Datasheet*, http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html, accessed May 8th, 2017.

[6] OpenMote: *OpenMote-CC2538*, http://www.openmote.com/hardware/openmote-cc2538-en.html, accessed May 8th, 2017.

[7] X. Vilajosana et al.: *OpenMote: Open-Source Prototyping Platform for the Industrial IoT*, In Proceedings of the 7th EAI International Conference on Ad Hoc Networks (AdHocNets), San Remo, Italy, September 2015.

[8] B. Ertl: *Data Aggregation using TinyIPFIX*, Bachelor Thesis, Chair for Network Architectures and Services, Computer Science Department, Technische UniversitÃ¤t München, München, Germany, April 2011.

[9] N. Tsiftes, J. Eriksson, A. Dunkels: *Low-Power Wireless IPv6 Routing with ContikiRPL*, In Proceedings of ACM/IEEE IPSN'10, Stockholm, Sweden, April 2010.

[10] TinyOS Alliance: *TinyOS*, http://www.tinyos.net/, accessed March 2017.

[11] A. Dunkels, B. Grönvall, 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, pp. 455-462, 2004.

[12] T. Winter et al.: *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, RFC 6550*, https://tools.ietf.org/html/rfc6550, March 2012.

[13] A. Dunkels: *Rime - A Lightweight Layered Communication Stack for Sensor Networks*, http://dunkels.com/adam/dunkels07rime.pdf, accessed May 8th, 2017.

[14] P. Levis et al.: TinsOS: An operating system for sensor networks, Ambient Intelligence, pp. 115-148, Springer, Berlin, 2005.

[15] A. Dunkels et al.: *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems*, In Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pp. 29-42, New York, USA, 2006.

[16] Wikipedia: *Type-length-value*, `https://en.wikipedia.org/wiki/Type-length-value`, accessed May 8th, 2017.

[17] A. Dunkels et al.: *Tunslip6 Application*, `https://github.com/contiki-os/contiki/blob/master/tools/tunslip6.c`, accessed May 8th, 2017.

[18] J. Romkey: *A nonstandard for transmission of IP datagrams over serial lines: SLIP, RFC 1055*, `https://tools.ietf.org/html/rfc1055`, accessed May 8th, 2017.

[19] D. Kopf: *rpl-border-router*, `https://github.com/lisgie/contiki/tree/master/examples/ipv6/rpl-border-router`, accessed May 8th, 2017.

[20] B. Claise et al.: *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information, RFC 7011*, `https://tools.ietf.org/html/rfc7011`, September 2013.

[21] A. Dunkels: *Contiki 3.0 changelog*, `https://github.com/contiki-os/contiki/issues/1197`, accessed May 8th, 2017.

[22] C. Schmitt et al.: *CoMaDa: An adaptive framework with graphical support for Configuration, Management, and Data handling tasks for wireless sensor networks*, In Proceedings of the 9th Conference on Network and Service Management, pp 211-218, October 2013.

[23] M. Zuniga, B. Krishnamachari: *Integrating Future Large-scale Wireless Sensor Networks with the Internet*, Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089, USA.

[24] J. Hui, P. Thubert: *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks, RFC 6282*, `https://tools.ietf.org/html/rfc6282`, September 2011.