Michael Spörk, BSc

IPv6 over Bluetooth Low Energy using Contiki

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dr. Carlo Alberto Boano

Institute for Technical Informatics

Graz, October 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Michael Spörk, BSc

IPv6 over Bluetooth Low Energy using Contiki

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Ass.Prof. Dr. Carlo Alberto Boano

Institut für Technische Informatik

Graz, Oktober 2016

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum

Unterschrift

Abstract

Bluetooth Low Energy (BLE) has gained popularity in research and industry over the past years because of its energy efficiency and reliability even under interference. Due to this popularity, the RFC 7668 standard was released in October 2015 that specifies the exchange of IPv6 packets over a BLE link layer and therefore enable BLE devices to connect to the Internet of Things. Although commercial devices implementing the RFC 7668 standard exist, no open source implementation of a compliant communication stack for constrained devices is currently available.

The major contribution of this thesis is the design of an IPv6 over BLE communication stack compliant to the RFC 7668 standard that fits the architecture of the Contiki OS, a popular operating system for constrained devices, and its implementation for the TI CC2650 SensorTag hardware platform. Furthermore, this thesis shows that the created IPv6 over BLE communication stack is interoperable with RFC 7668 compliant devices, and provides BLE nodes with network connectivity. A first comparison of IPv6 over BLE to IPv6 over IEEE 802.15.4 using the same hardware platform shows that the IPv6 over BLE stack requires less memory, has a lower communication overhead, and provides better reliability under interference than IPv6 over IEEE 802.15.4 at the cost of less energy efficiency.

Kurzfassung

Bluetooth Low Energy (BLE) hat aufgrund seiner Energieeffizienz und Störungssicherheit immer mehr Popularität in Forschung und Industrie gewonnen. Im Oktober 2015 wurde deshalb der RFC Standard 7668 veröffentlicht, der den Austausch von IPv6-Paketen über eine BLE-Funkverbindung spezifiziert und somit BLE-Geräten eine Verbindung zum Internet der Dinge ermöglicht. Obwohl bereits kommerzielle Geräte verfügbar sind, die den RFC 7668 Standard unterstützen, gibt es noch keine Open Source Implementierung eines RFC 7668-konformen Kommunikationsstacks für Geräte mit begrenzten Ressourcen.

Diese Masterarbeit präsentiert das Design eines RFC 7668-konformen Kommunikationsstacks für Contiki, ein weitverbreitetes Betriebssystem für Geräte mit begrenzten Ressourcen, und die Implementierung dieses Kommunikationsstacks für die CC2650 SensorTag Hardware von Texas Instruments. Der erstellte Kommunikationsstack für IPv6 über BLE-Funkverbindungen ist kompatibel mit Geräten, die den RFC 7668 Standard erfüllen, und ermöglicht es BLE-Geräten mit anderen Netzwerkteilnehmern zu kommunizieren. Der erste Vergleich zwischen IPv6 über BLE-Verbindungen und dem bestehenden IPv6 über IEEE 802.15.4-Verbindungen zeigt, dass der neue Kommunikationsstack weniger Speicher benötigt, weniger Kommunikations-Overhead aufweist und zuverlässiger ist als IPv6 über IEEE 802.15.4-Verbindungen, zum Preis von geringerer Energieeffizienz.

Acknowledgments

This Master Thesis was written during the year 2016 at the Institut for Technical Informatics at Graz University of Technology.

First and foremost, I would like to thank my supervisor Carlo Alberto Boano for his excellent support during my work on this Master Thesis. Without his supervision and feedback, this thesis would not have been possible at its current scope. Moreover, I thank Markus Schuß for his help and support during this thesis.

I would like to thank my partner Verena for her aid, motivation and understanding. Finally, I thank my family, especially my parents, for their education, backing and support. Without them, I would certainly not be at this point in my life.

Graz, October 2016 Michael Spörk

Danksagung

Diese Diplomarbeit wurde im Jahr 2016 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Zuallererst möchte ich meinem Betreuer Carlo Alberto Boano für seine hervorragende Unterstützung während der Durchführung meiner Masterarbeit danken. Ohne seine Betreuung und sein Feedback wäre diese Masterarbeit sicher nicht in diesem Umfang möglich gewesen. Zudem möchte ich auch Markus Schuß für seine Hilfe und Unterstützung während dieser Arbeit bedanken.

Ich möchte mich bei meiner Partnerin Verena für ihren Beistand, ihre Motivation und ihr Verständnis danken. Abschließend danke ich meiner Familie, insbesondere meinen Eltern, ohne deren Erziehung, Rückhalt und Unterstützung ich sicher nicht an diesem Punkt meines Lebens stehen würde.

Graz, im Oktober 2016 Michael Spörk

Contents

List of Figures

List of Tables

Abbreviations

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks

ATT Attribute Protocol

BLE Bluetooth Low Energy

CCA Clear Channel Assessment

CoAP Constrained Application Protocol

CPU Central Processing Unit

GAP Generic Access Profile

GATT Generic Attribute Profile

HAL Hardware Abstraction Layer

HCI Host Controller Interface

HTTP Hypertext Transfer Protocol

IEEE Institure of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

ICMPv6 Internet Control Message Protocol version 6

IoT Internet of Things

IP Internet Protocol

IPHC IP Header Compression

IPSS Internet Protocol Support Service

IPv6 Internet Protocol version 6

ISM Industrial, Scientific and Medical

L2CAP Logical Link Control and Adaptation Protocol

MAC Media Access Control

MCU Microcontroller Unit

MTU Maximum Transmission Unit

NA Neighbor Advertisement

NS Neighbor Solicitation

OS Operating System

RA Router Advertisement

RAM Random Access Memory

RDC Radio Duty Cycling

RFC Request for Comments

ROM Read-only Memory

RPL Routing Protocol for Low power and Lossy Networks

RS Router Solicitation

SM Security Manager

SMP Security Manager Protocol

SoC System on Chip

TCP Transmission Control Protocol

TI Texas Instruments

UDP User Datagram Protocol

USB Universal Serial Bus

WPAN Wireless Personal Area Network

Chapter 1

Introduction

Connecting everyday objects to the Internet and hence creating an "Internet of Things" has received much attention in recent years. Internet of Things (IoT) applications range from personal health and fitness monitoring over home automation to industrial applications such as smart grids. Although the application requirements are very different, most objects and devices in these applications are required to operate for several years on constrained power sources like coin cell batteries.

In addition to the requirement to consume as little energy as possible, some objects do not have a tethered data communication and need to use wireless communication methods for data exchange. Since such wireless communication methods are quite energy demanding, wireless communication is accountable for a significant part of a wireless devices' consumed energy. To minimize the energy consumed for communication, low power wireless communication methods such as duty cycling [17] are widely used in untethered objects and devices of the IoT.

A major research focus in the past years was to standardize and optimize the exchange of Internet Protocol version 6 (IPv6) packets over the wireless low power link layer technology IEEE 802.15.4 using the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) adaptation layer defined by the RFC 4944 [32], RFC 6282 [29] and RFC 6775 [46] standards. With the RFC 7668 standard [34] released in October 2015, which defines the exchange of IPv6 packets over Bluetooth Low Energy (BLE), an additional link layer for Internet of Things applications was proposed. BLE promises several advantages over IEEE 802.15.4, such as lower energy consumption, built-in security and privacy features. Also, BLE is widely adopted in consumer devices like smartphones, tablets and laptop. These devices could be used to provide Internet access to embedded systems that communicate using IPv6 over BLE or could extend the range of IPv6 over BLE networks [1].

1.1 Problem statement

IPv6 over Bluetooth Low Energy is already used in commercial devices. For example, Nordic Semiconductors already markets chips with IPv6 over BLE support [37]. However, although commercial products supporting IPv6 over BLE are existing, no open source implementation of the RFC 7668 standard [34] for constrained devices is currently available.

An open source implementation of the RFC 7668 standard would help in supporting further research on IPv6 communication over BLE and trigger the design of new strategies and optimizations for example the use of BLE advertisement channels to broadcast small IPv6 packets. The latter would enable the infrequent exchange of short IPv6 data while possibly consuming less energy then IPv6 over Institure of Electrical and Electronics Engineers (IEEE) 802.15.4 and existing IPv6 over BLE.

One example on how an open source implementation of a communication stack leads to improvements of existing or even novel RFC standards is the Contiki Operating System (OS) [18, 44]. The Contiki OS is an open source operating system for constrained devices that provides a full IP network stack with UDP, TCP and HTTP support over an IEEE 802.15.4 link layer. Since its creation, Contiki supported further research on IPv6 over IEEE 802.15.4 communication that led to novel media access protocols, such as ContikiMAC, or routing protocols such as the Routing Protocol for Low power and Lossy Networks (RPL).

I therefore aim to implement an open source IPv6 over BLE stack for the Contiki OS such that further research on IPv6 over BLE can be done and optimized or novel ways to exchange IPv6 packets over BLE may be created. The main challenge of adding BLE support to the Contiki OS is that the Contiki communication stack is highly interwoven with the currently used link layer IEEE 802.15.4. Thus, adding BLE support means that the new functionality not only needs to adhere to the Bluetooth specification [3], but also needs to fit into the existing architecture of the Contiki OS and its communication stack. Furthermore, the developed stack should be interoperable with other IPv6 over BLE implementations that are compliant to the RFC 7668 standard [34] and efficient in both its memory usage and power consumption.

1.2 Thesis contribution

This Thesis presents an implementation of the RFC 7668 standard [34] for the Contiki OS with the following features:

- fully open source implementation of the Bluetooth Low Energy host;
- fully interoperable communication stack with IPv6 over BLE devices compliant to the RFC 7668 standard [34];
- fully compatible with the Contiki OS architecture, making sure that the programmer can use the same application code for BLE and IEEE 802.15.4 applications.
- Contiki applications using IPv6 communication are agnostic to the radio technology used (i.e., whether IEEE 802.15.4 or BLE).

This Thesis further presents an evaluation of the energy consumption of the IPv6 over BLE communication stack executed on the TI SensorTag platform, and compares the energy efficiency of the IPv6 over IEEE 802.15.4 communication stack and IPv6 over BLE. Finally, I present the robustness of IPv6 over BLE under interference and compare it with the robustness of 6LoWPAN over IEEE 802.15.4 under the same conditions and show that although the current implementation of the IPv6 over BLE communication stack is

less energy efficient than the existing IPv6 over IEEE 802.15.4 stack, it very robust and provides packet reception rates of 100% even under heavy WiFi interference.

1.3 Limitations

The communication stack implemented in this Thesis has the following limitations:

1.3.1 Single hardware platform

The presented implementation of the RFC 7668 standard is designed to be generic and fairly easy to port to other BLE supporting hardware. However, due to time constraints the current implementation only supports the SensorTag platform of Texas Instruments.

Section 4.3.3 provides a guide on porting the IPv6 over BLE communication stack implemented for the SensorTag platform to other hardware platforms supporting the Host Controller Interface (HCI) of BLE.

1.3.2 Bluetooth version 4.1

The used SensorTag hardware platform only supports Bluetooth Low Energy communication according to the Bluetooth Specification version 4.1 [3] and not as defined by the newer Bluetooth Specification version 4.2 [5]. The only difference between these two Bluetooth versions that is relevant for this Thesis it that version 4.2 supports BLE data packets with a length of up to 255 bytes while version 4.1 limits the BLE data packet length to 31 bytes.

1.3.3 Limited BLE stack

The BLE stack implemented in this Thesis is no full BLE implementation as specified by the BLE standard [3], but only includes the features needed for IPv6 over BLE on node devices. Functionalities like the Generic Access Profile (GAP) (defines general modes, generic procedures and used terminology for BLE devices), Security Manager (SM) (provides procedures for authentication and encryption between BLE devices), Generic Attribute Profile (GATT), Attribute Protocol (ATT) (both used to exchange device specific attributes and parameters between BLE devices) and parts of Logical Link Control and Adaptation Protocol (L2CAP) that are part of a full standard BLE stack are not supported by the implementation presented in this Thesis.

Nevertheless, the implemented communication stack provides BLE advertisement and is able to exchange data on the BLE data channels when a BLE connection has been established.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces BLE and discusses how it can be used in IoT applications for wireless data exchange using the RFC 7668 standard [34]. It also summarizes the benefits of using IPv6 over BLE in the IoT and introduces the reader to the Contiki OS and the hardware used in this thesis. Chapter 3

lists the already existing BLE communication stacks and evaluates these stacks according to the requirements of this thesis. Related studies that examine the performance of BLE and compares it to other existing wireless communication technologies are also shown in Chapter 3. Chapter 4 presents the IPv6 over BLE communication stack implemented for this thesis, discusses the design decisions and implementation details made during its development, and provides a guide for porting the communication stack to other hardware platforms. The performance of this implemented IPv6 over BLE communication stack is evaluated in Chapter 5 and concluded in Chapter 6. Chapter 7 provides an overview of possible improvements to the presented communication stack and possible further research on IPv6 over BLE.

Chapter 2

Background

This chapter introduces the reader to Bluetooth Low Energy (BLE) technology and all relevant aspects and communication standards used in or related to this Thesis. Section 2.1 introduces Bluetooth and BLE as wireless communication technologies and discusses the motivation behind creating BLE and the differences between BLE and classic Bluetooth. Besides describing the BLE communication stack, this section also enumerates possible advantages of BLE as a link layer in the IoT.

The benefits of using the IPv6 in IoT applications is highlighted in Section 2.2. After listing the advantages of using IPv6 on top of low-power and lossy link layer technologies, the 6LoWPAN adaptation layer over 802.15.4 [29, 46] and the 6LoWPAN adaptation layer for BLE devices [34] are presented.

Section 2.3 gives an overview on the Contiki Operating System, a popular OS for constrained devices in the IoT that I use to implement the work presented in this Thesis. The last section of this chapter, Section 2.4, details on the hardware platforms used in this Thesis.

2.1 Bluetooth Low Energy (BLE)

Bluetooth is a standard for wireless communication which was introduced in 1999 by an alliance of companies led by Ericsson [2]. The primary goal of Bluetooth was to create and establish a global standard for wireless data exchange between constrained devices in close proximity to each other. The main purpose of this new standard was to replace cables, especially serial data cables between devices [23].

Bluetooth is currently available in almost every smartphone, tablet computer and notebook, and has a wide variety of applications such as wireless mouse and keyboard, wireless headset, and file exchange between mobile devices.

2.1.1 Enhancing classic Bluetooth

In June 2002 a new IEEE standard based on the Bluetooth Specification v1.1 was approved and named 802.15.1 [35]. The latter specified Bluetooth v1.1 as a new standard for Wireless Personal Area Networks (WPANs). The standard proposed a variety of use cases for the wireless communication technology, but this wide spectrum of applications came with several disadvantages. The resulting communication stack is indeed more complex and

has a poorer performance than communication stacks of comparable technologies such as IEEE 802.15.4 [3, 9, 31] as shown in Figure 2.1. Adaptive frequency-hopping, which was added in the Bluetooth Specification v1.2, improved the performance of Bluetooth but made the specification even more complex.

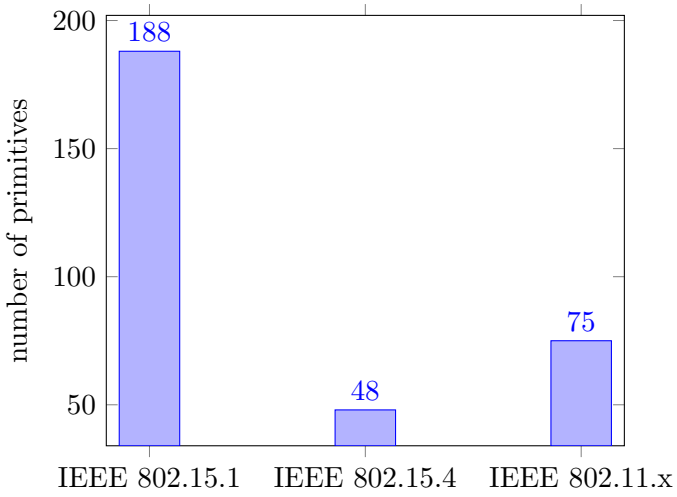

Figure 2.1: Number of primitives of IEEE 802.15.1 (classic Bluetooth), IEEE 802.15.4 and IEEE 802.11.x (WiFi) (adapted from [30])

For this reason, Bluetooth was redesigned and the new design was merged into the Bluetooth core Specification version 4.0. This new system is called Bluetooth Low Energy (Bluetooth LE or BLE) and is marketed as Bluetooth Smart.

The main design objective of BLE was to create a wireless technology for short range communication with ultra-low power consumption. The link setup time between two BLE devices was shortened which makes it viable to infrequently exchange small chunks of data over a reliable and secure link layer. Besides the power consumption, the low cost of BLE hardware and the reduces complexity and small size of the BLE communication stack were also goals of BLE [23, 24].

Since the Bluetooth Specification v4.0 were published in 2010, Bluetooth has two different systems: the classical Bluetooth and Bluetooth LE. These two systems are not interoperable [2]: devices may implement both of these Bluetooth systems, but can only use one of them at a certain point in time.

2.1.2 Bluetooth LE key features

I now list the key features of the BLE standard in detail [23]:

Globally available

BLE operates in the 2,4 GHz Industrial, Scientific and Medical (ISM) frequency band. By using this ISM band, Bluetooth LE can be used globally without any license fee. However, this ISM band is shared with several other wireless technologies such as Wireless LAN (Wi-Fi) and IEEE 802.15.4, and radio interference from other devices is hence possible.

Furthermore, other devices and home appliances such as microwave ovens can emit noise and disturb communications in those frequencies. Such disturbances and interference may cause connection problems or even connection loss. BLE uses adaptive frequency-hopping as a method to counteract interference from other devices and appliances and thus provides a reliable link between BLE devices.

Fast connection setup

The BLE standard specifies three radio channels that are only used for advertising data and communication setup. A BLE device needs to check only those radio channels for connection requests, instead of scanning each possible channel for any incoming connections as in classic Bluetooth. By using these dedicated channels, a connection between two Bluetooth LE devices can be set up in less than three milliseconds.

Mostly-Off Technology

The default state of Bluetooth LE devices is „switched-off“. In this state the radio is disabled and the device consumes very little energy. BLE devices only enable the radio communication if there is data to communicate. There is no need for polling between two connected devices.

Reduced functionality

The functionality of the Bluetooth LE communication stack is significantly reduced in comparison to the communication stack of classic Bluetooth. BLE does not support features like Scatternets, voice channels, continuous polling or Master/Slave role switch that are available in classic Bluetooth. This reduced functionality simplified the complexity of the BLE communication stack. As a result, the communication stack of BLE has a much smaller static and dynamic memory footprint then classic Bluetooth.

Optimized operations

While central devices (e.g., Bluetooth scanner, laptop) are normally connected to a powerful battery or even a continuous power supply, peripheral devices (e.g., wearables, sensors) typically operate on constrained battery power. BLE minimizes by design the power consumption of peripheral devices, while compensating for this optimization with a less power-optimized central device.

2.1.3 BLE communication stack

The BLE communication stack is divided into seven layers. As Figure 2.2 shows, the communication stack is split into two main parts: the Controller and the Host.

The Controller is typically implemented in a small System on Chip (SoC) and comprises the two lower layers of the communication stack: the Physical Layer and the Link Layer. The host runs on the application processor that implements the five upper layers: the Logical Link Control and Adaptation Protocol (L2CAP), the Attribute Protocol (ATT), the Generic Attribute Profile (GATT), the Security Manager Protocol (SMP) and the Generic Access Profile (GAP) of the BLE communication stack.

The Host and the Controller may communicate with each other via the standardized Host Controller Interface (HCI) [3, 22, 23].

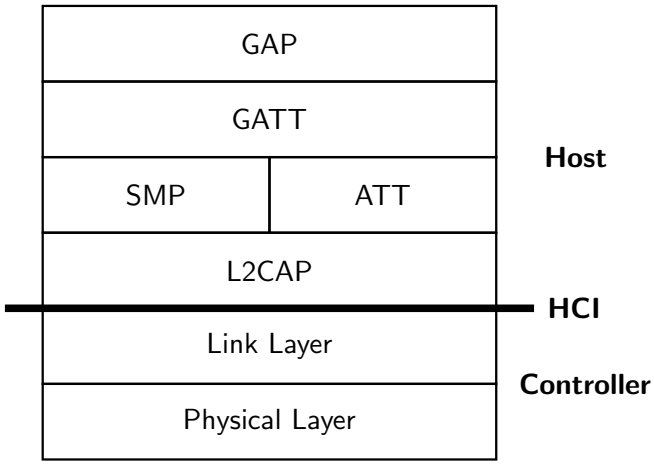

Figure 2.2: BLE communication stack (adapted from [23]).

Physical Layer

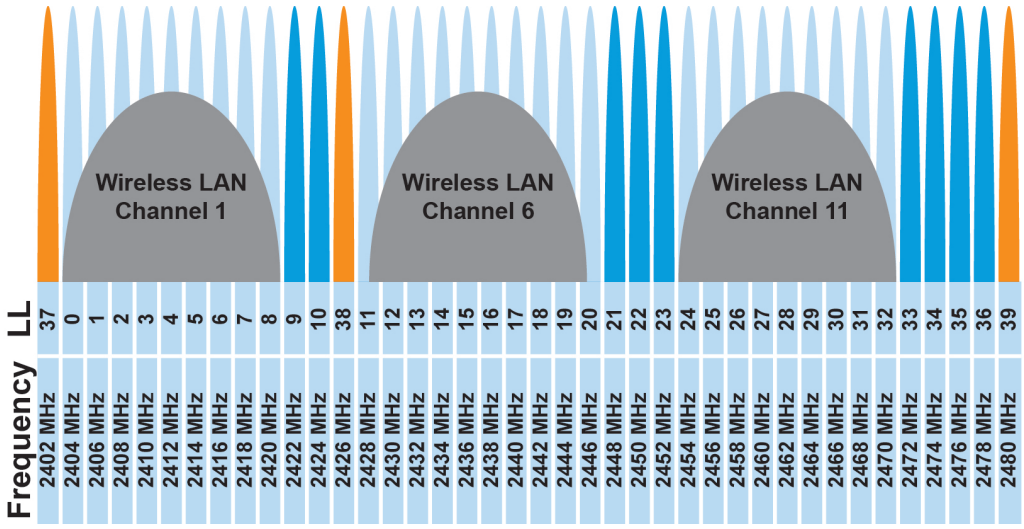

Figure 2.3: physical channels of BLE overlapping with the three most popular Wireless LAN channels (http://www.connectblue.com/press/articles/shaping-the-wireless-future-with-low-energy-applications-and-systems/).

Bluetooth LE operates in the globally unlicensed 2,4 GHz band that is divided into 40 separate BLE radio channels with 2 MHz channel spacing. All channels use GFSK (Gaussian Frequency Shift Keying) for data modulation and support a physical data rate of 1 Mbps. Ordinary BLE devices have a transmission distance of a few tens of meters.

Out of those 40 BLE radio channels, three channels (channel number 37, 38 and 39)

are advertising channels. These advertising channels are only used for Bluetooth LE device discovery, connection setup and data broadcasting. The radio frequencies of the advertising channels are selected to have minimal overlap with the most popular Wireless LAN channels to counteract interference and therefore ensure short device discovery and connection setup times. Figure 2.3 shows the 40 BLE radio channels overlapping with the three most popular Wireless LAN channels (Wireless LAN Channel 1, 6 and 11).

The remaining 37 radio channels are data channels. These channels are used only for transmitting data between devices that already have an established connection. An adaptive frequency hopping mechanism is used on top of the data channels to minimize the negative effects of interference and other wireless propagation issues.

BLE devices may only have transmission or receiving capabilities to broadcast their measured data or receive configuration data, respectively. This can help in further decreasing device complexity and cost.

Link Layer

BLE devices are identified via 48-bit device addresses. A device address can either be a public address (a globally unique 48-bit identifier), or a random address (a randomly created 48-bit identifier that may change periodically). Random device addresses are a privacy feature to hide the devices' public address and to protect against device tracking.

The simplest way to transmit data via Bluetooth LE is to broadcast data via an advertising event. Advertising events are periodical events in which an advertiser broadcasts advertisement packets to all BLE devices in its proximity. Devices listening to such advertisement packets are called scanners. Advertisement packets, that carry the transmitted data as a payload, are sequentially sent on all three advertising channels during an advertising event. Data transmitted using BLE advertisement may only be sent unidirectionally from advertiser to scanners and is not acknowledged by any of the scanners.

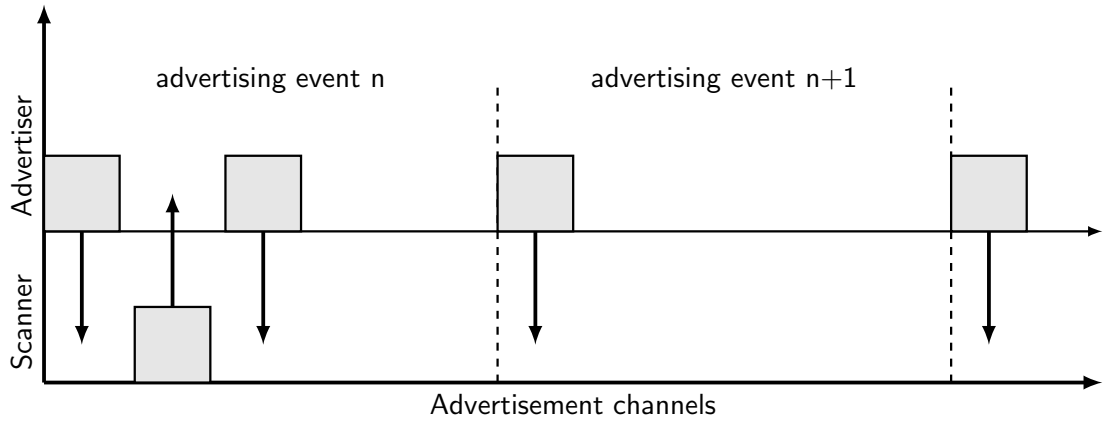

Figure 2.4: Two consecutive advertising events: advertising event n shows one advertisement, one scan request and a corresponding scan response, advertising event n+1 shows an advertising event without scanning (adapted from [23]).

Figure 2.4 shows consecutive advertising events. At the start of each advertising event the advertiser sends out an advertisement packet. If any scanner device receives the

advertising packet it sends out a scan request to which the advertiser may answer with a scan response and waits for the next advertising event (shown in advertising event n). When the advertiser does not receive any scan request it simply waits for the next advertising event without advertising any additional data (shown in advertising event n+1).

To exchange data bidirectionally, two devices need to establish a connection with each other. The connection setup is performed on the advertising channels. To start a connection, one device advertises that it supports connections. An initiator receiving such an advertisement transmits a connection request to create a connection between the initiator and the advertiser. During this connection setup, the initiator provides the advertiser with information for the adaptive frequency hopping (which data channels to use) and timing parameters for the established connection (e.g., when the slave should wake up). After successfully establishing the connection, both devices are in the connected link layer state, where the previous initiator is the master and the previous advertiser the slave of the connection.

The exchange of data between master and slave is called a connection event. At the beginning of a connection event (also called anchor point), the master sends a data packet to the slave, to which the slave has to respond. After these first packets of the connection event, master and slave may exchange further data until the connection event is closed. All Link Layer connections provide packet acknowledgment and flow control to the upper layers of the BLE stack. This ensures that all packets sent using connection events are transmitted correctly and in order. The adaptive frequency hopping algorithm selects a new data channel at the beginning of each connection event (anchor point) and both devices stay on this data channel until the connection event ended.

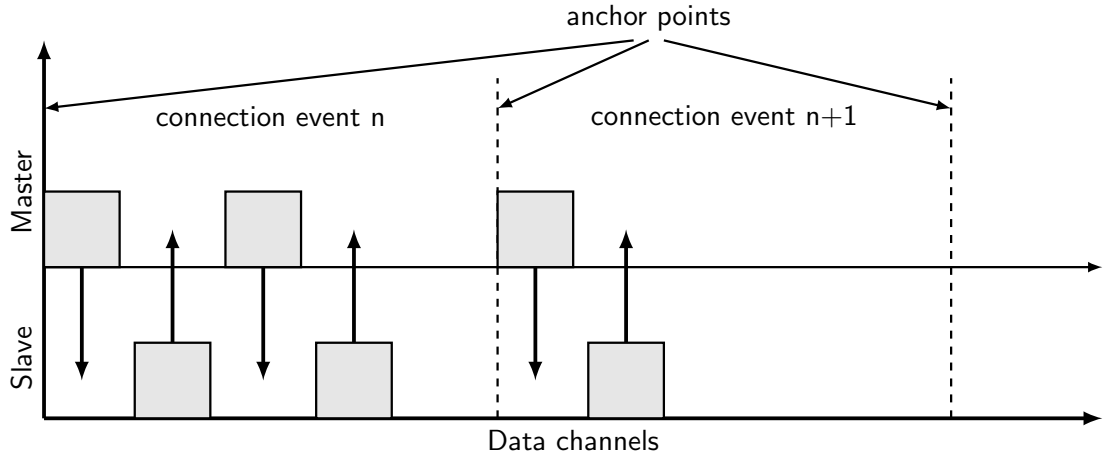

Figure 2.5: Two consecutive connection events where several BLE data packets are exchanged between master and slave (adapted from [23]).

Figure 2.5 shows two consecutive connection events. During connection event n the master and slave exchange several BLE data packets. Connection event n+1 only the minimal exchange between master (sending a packet at each anchor point) and slave (required empty response to the first master packet).

Logical Link Control and Adaptation Protocol (L2CAP)

The L2CAP layer is the interface between the higher layer protocols and the lower layers of the BLE communication stack. This layer is responsible for multiplexing the higher layer protocols, so that each protocol can use the shared logical link safely. The L2CAP also provides packet segmentation and packet reassembly for the upper layers in the communication stack.

Attribute Protocol (ATT)

The ATT layer provides functionality for discovering, retrieving, and setting attributes (e.g., device name, device model number, sensor values) of a remote BLE device. The communication of the Attribute Protocol follows a client server model. A server provides a set of attributes that can be discovered and may be set or read. The corresponding client can interact with those attributes. The client and server roles can be chosen independently from the master or slave role of the link layer.

Generic Attribute Profile (GATT)

An application on a BLE device can use the GATT protocol to retrieve exposed services and characteristics of a connected remote device (e.g., a remote device can expose a temperature service and the corresponding characteristics of this temperature service, like temperature and unit). To exchange the data of the services and characteristics, GATT uses the Attribute Protocol.

Security Manager (SM)

The Security Manager provides functionality for encrypting and authenticating data packets using a 128-bit AES block cipher. To enhance the privacy of a BLE device, the Security Manager additionally provides methods for generating a random device address.

Generic Access Profile (GAP)

The Generic Access Profile is the highest layer in the BLE communication stack. It defines the role of a device and the modes and procedures used for device discovery, service discovery, connection setup, and security. These definitions ensure interoperability between BLE devices from different manufacturers.

The GAP defines four possible device roles with the corresponding requirements to the controller and BLE procedures. A broadcaster periodically transmits data using advertising events. An observer receives advertising events from broadcasters.

A device acting as central device is able to initiate connections to several peripherals and acts as the master. The peripherals are the slaves and can only accept one connection to a single master.

2.1.4 BLE and the Internet of Things

Although classic Bluetooth had several drawbacks when used in WPANs, sensor networks and Internet of Things applications (especially in comparison to competing technologies

such as IEEE 802.15.4), Bluetooth LE may instead exhibit a number of advantages over IEEE 802.15.4.

Several measurements [16, 40] have indeed shown that BLE has a lower energy consumption than IEEE 802.15.4, especially on peripheral devices. Another advantage of BLE is that it is already widely adopted in wireless devices like smartphones, smartwatches, fitness and medical devices, which means that smartphones may act as routers and use their mobile data and Wi-Fi connections to provide internet access to other BLE devices [1].

2.2 IPv6 over low-power lossy links

As discussed in 2.1.4, Bluetooth LE has several advantages over currently used link layer protocols like IEEE 802.15.4 in Internet of Things applications where small amounts of data is periodically exchanged using constrained devices (e.g., smart sensors, medical devices). But to interoperably communicate with devices in the IoT, BLE devices cannot simply use their ordinary BLE packets. The most common way to exchange data in the IoT is the Internet Protocol version 6 [45].

2.2.1 Internet Protocol version 6 (IPv6)

The benefits of using IPv6 for connecting Smart Objects to the Internet have already been discussed in detail in [39] and [45]. This section provides a short summary of the benefits of IPv6 in the context of embedded devices.

Interoperability

IPv6 is the standard protocol for the Internet and enables devices to communicate with any other device on the Internet without the need for additional hardware or software. For the deployment in the Internet, IPv6 operates on top of very different link layer protocols, such as Ethernet, Wireless LAN and and fiber-optic communication. Although devices use different link layer technologies, they are able to exchange data using IPv6.

Evolvable and versatile architecture

The IP architecture is designed to use the end-to-end principle, which states that all application layer functionality is implemented in the end points of the network. The network only transports data to the end points, but does not have any application-level intelligence. This end-to-end principle makes IPv6 evolvable and versatile, because the network functionality does not change even if applications are added, removed, extended, or updated.

Scalability and stability

The scalability and stability of IP has been proven by the Internet, where IP has shown to be scalable and stable in a global deployment. With its 128-bit addresses, IPv6 supports up to approximately $340 * 10^{36}$ connected devices.

Configuration and Management

There are many existing protocols and network tools, that provide advanced configuration and network management functionality, such as DHCP and SNMP. By communicating with IPv6, those tools can be used to configure and manage all deployed devices.

Small footprint

Embedded devices need to be low energy, physically small, and cost effective. These requirements translate to a severe memory and code complexity constraints. Although the IP architecture was previously thought of as heavyweight, several implementations exist that only need few kilobytes of RAM and ROM.

2.2.2 6LoWPAN adaptation layer

The 6LoWPAN adaptation layer provides methods to compress ordinary IPv6 traffic to fit the needs of constrained objects. This adaptation layer was designed to make IPv6 traffic over IEEE 802.15.4 links possible. It defines standards for IPv6 header compression [29] and neighbor discovery [46] on low power link layer technologies. By implementing these standards, a 6LoWPAN layer is introduced between the network and the link layer as shown in Figure 2.6, which reduces the protocol overhead of IPv6 and makes IPv6 communication over IEEE 802.15.4 links possible.

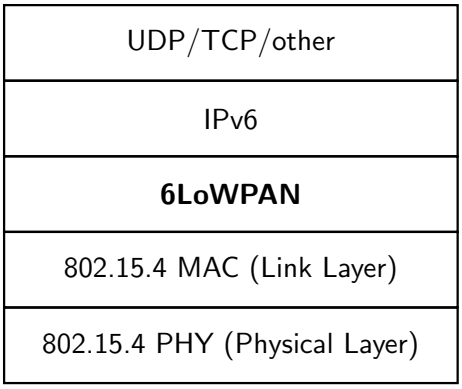

Figure 2.6: 6LoWPAN network stack (adapted from [39]).

2.2.3 6LoWPAN adaptation layer for BLE devices

Not all of the functionality specified in the classical 6LoWPAN standards is needed for implementing IPv6 over BLE, so the Internet Engineering Task Force (IETF) proposed a standard RFC 7668 [34] that specifies the 6LoWPAN layer that enables IPv6 communication on BLE link layers. As shown in Figure 2.7, the 6LoWPAN for BLE layer is located on top of the BLE L2CAP layer and below the IPv6 layer.

While other 6LoWPAN standards specify functions for data fragmentation and reassembly, RFC 7668 does not include such functions. The fragmentation and defragmentation of data is already handled in the L2CAP layer of the BLE communication stack.

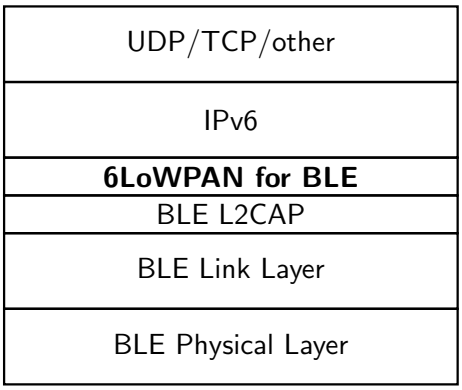

Figure 2.7: IPv6 on the BLE communication stack (adapted from [34]).

Network topology

Figure 2.8 shows the simplest case of a subnet which exchanges data using IPv6 over BLE links. This simple subnet consists of six node devices, each of these nodes acts as a BLE peripheral device, and one single border router, acting as a BLE central device. Each of the node devices is connected to the border router via an individual BLE data connection [34].

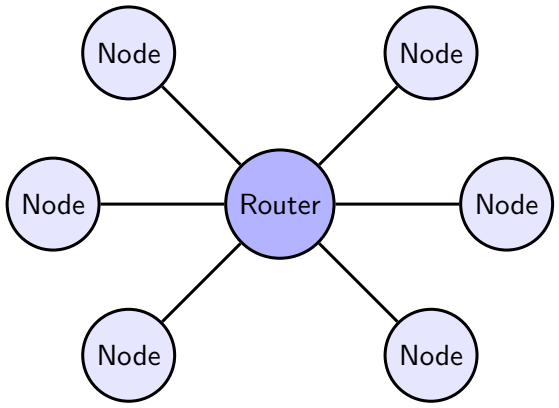

Figure 2.8: simple IPv6 over BLE subnet with 6 node devices and a single border router; nodes in the subnet may exchange data but cannot communicate with devices outside of the subnet (adapted from [34]).

A convention of IPv6 states that IPv6 subnets should span over a single link layer [15], yet a multilink model, where each node has a separate link to the router, has been chosen to make the network topology of IPv6 over BLE more suitable for constrained devices. This multilink model limits the use of link-local communication only to individual BLE connections, link-local communication between two node devices is not possible. Therefore nodes connected to the same border router have to use the shared prefix to exchange data. The border router in IPv6 over BLE networks acts as an IPv6 router and not as a link-layer switch. [34]

The Figure 2.9 shows a more typical scenario for IoT applications, where the node

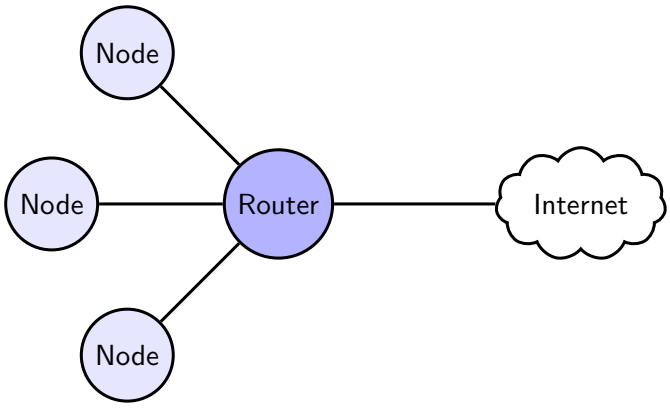

Figure 2.9: typical IPv6 over BLE subnet connected to the Internet; nodes may exchange data within the subnet or interact with devices on the Internet (adapted from [34]).

devices have access to the Internet. In this scenario all nodes in the subnet share a 64-bit IPv6 prefix. The border router in the subnet is responsible creating the BLE connection to each of the nodes, for distributing the shared IPv6 prefix to all connected node devices and for forwarding IPv6 packets from and to the Internet.

Stateless IPv6 address autoconfiguration

Devices generate their link-local IPv6 address based on the 48-bit Bluetooth device address. This generation is performed according to IEFT standards RFC 7136 [11] and RFC 4291 [25], where a 64-bit IID (interface identifier) is formed and an IPv6 address prefix is appended, respectively.

Non-link-local IPv6 addresses do not embed the Bluetooth device address in their 64-bit IID by default. Instead, cryptographic, hash-based, or other privacy enabling methods are used to create the IID. If the Bluetooth device address is known to be a random device address, this random address can be embedded in the IID as stated in RFC 7136 [11]. The prefix of the non-link-local addresses is provided by the router of the network.

Neighbor Discovery

The IETF RFC 6775 [46] defines how neighbor discovery in low power WPANs is performed. Since BLE only supports star network topologies, only functions regarding these topologies are considered for IPv6 over BLE.

A device registers its non-link-local address at the router using Neighbor Solicitation (NS) messages with the Address Registration Option (ARO) set. The router confirms the devices‘ address by sending a Neighbor Adversitement (NA). Link-local addresses must not be registered at the router.

A router exposes itself to a connected device by sending Router Advertisements (RA) and a device can ask for a router by sending Router Solicitations (RS).

Header compression

IPv6 over BLE compresses IP headers according to RFC 6282 [29]. Additionally to the header compression techniques defined in the RFC, the star topology of BLE networks can be exploited to compress device addresses even further. Figure 2.10 shows a compressed IPv6 packet which is compressed using the IPHC scheme.

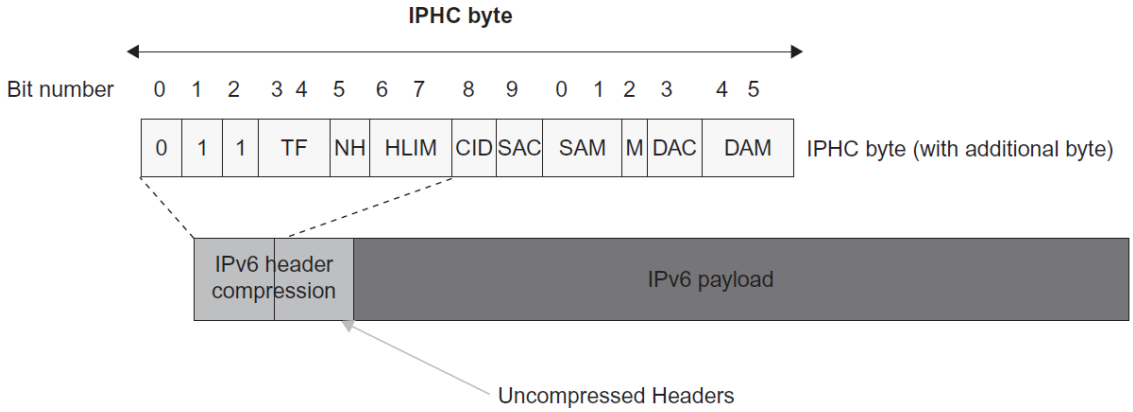

Figure 2.10: Compressed IPv6 packet with IP Header Compression (IPHC) [45].

When a device sends a packet to the router using the link-local address, the source address is fully elided. If the address context was sent by the router and the destination address matches such a context, the destination address is fully or partially elided and the header compression flags (DAC, DAM) are set accordingly.

If a router sends a packet to a connected device with its link-local address based on the Bluetooth device address, the IID of the source address is elided. The source address prefix is also elided, if a related address context has been set up. The link-local destination address (which is based on the Bluetooth device address or related to an already set up address context) is elided too. If any address is elided, the header compression flags (SAC, SAM, DAC, DAM) are set according to the RFC 6282 standard [29], which specifies the standards for IP header compression.

2.3 The Contiki Operating System

The Contiki OS is a lightweight, open source operating system for constrained devices. It is written in the C programming language and is marketed as the “Open Source OS for the Internet of Things” [18, 44].

Because Contiki is designed for tiny systems, the OS operates on a few kilobytes of memory, less than 10kB RAM and 30kB ROM, while providing a full IP network stack with UDP, TCP and HTTP support. In addition to the IP stack, which is fully certified under the IPv6 Ready Logo, Contiki also provides low power communication protocols such as the 6LoWPAN adaptation layer, RPL multi-hop routing and, CoAP [44].

Contiki currently supports three primary hardware platforms: the Texas Instruments CC2538, the Texas Instruments SensorTag and the Texas Instruments CC2650, as well as

16 other platforms ranging from 8051-powered SoC over the Tmote Sky platform using a Texas Instruments CC2420 radio to various ARM devices [43].

2.3.1 Kernel and Protothreads

Usually memory constrained platforms use event-driven mechanisms to provide concurrency. Such event-driven mechanisms have a much lower memory consumption than multi-threaded systems, since event-driven systems do not need an individual stack for each of the created threads because all processes in an event-driven system share the same stack. Another advantage of event-driven systems is that locking mechanisms are generally not needed because multiple event handlers are not able to run concurrently. Although such event-driven systems provide advantages for platforms with limited memory, not all programms can be easily expressed in an event-driven way and developing an event-driven application is usually more complicated than creating a multi-threaded application [18].

The Contiki OS combines an event-driven system with preemptible threads using an event-driven kernel and a user library providing multi-threading functionality. The Contiki kernel dispatches events to running processes by calling the polling handler of the respective process. Each called polling handler runs to completion and cannot be preemptively stopped by the kernel [18].

Contiki provides system libraries that may be optionally linked with programs and implement functionality additional to the basic event handling in the kernel. Amongs others the system libraries implement multi-threading, memory management and communication support [18, 44].

2.3.2 Contiki network stack

The network stack of Contiki (shown in Figure 2.11) consists of four network layers: the Radio layer, the Radio Duty Cycling (RDC) layer, the Media Access Control (MAC) layer and the Network layer.

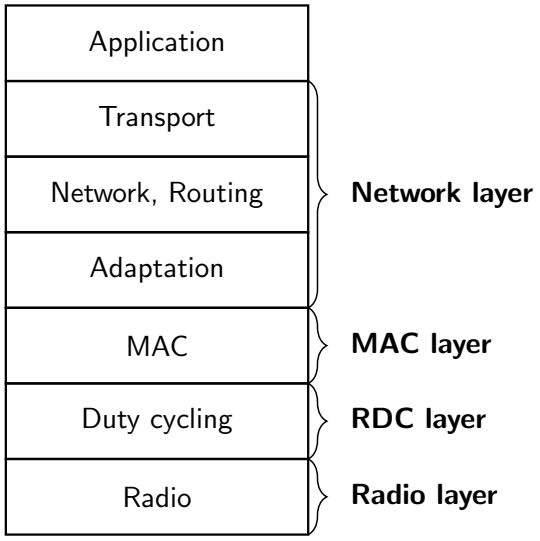

Figure 2.11: Network stack of the Contiki OS.

The radio layer implements the functionality of sending and receiving radio packets. It provides a method to send a packet and triggers an interrupt if data was received. The Clear Channel Assessment (CCA) is typically performed in this layer as e.g., implemented for the Tmote Sky and its Texas Instruments CC2420 radio.

The RDC layer handles the duty cycling of the radio. It provides methods for sending a single or a list of data packets over the radio and also notifies the upper layer whenever a list of data packets or an individual packet was received. This layer is also responsible for converting packet attributes (e.g., sender address, receiver address and sequence numbers) to link layer headers, which can be transmitted over the radio. An example for an implementation of the RDC layer is ContikiMAC.

The MAC layer implements the media access control of the network stack. This layer provides methods for sending packets to and receiving packets from neighbor devices. Any additional security operations on the communication data are executed in this layer as implemented in the CSMA implementation in Contiki.

The layer on top of the network stack of Contiki is the network layer. This layer is responsible for the data adaptation (IPv6 header compression, data fragmentation and reassembly). Additionally to the data adaptation, this layer also includes network and routing functionality as well as transport and application logic.

The existing network stack implementation supports IPv6 or RIME communication over an IEEE 802.15.4 radio layer and provides several different existing RDC and MAC layer implementations to fit the needs of various applications. RIME communication is used in wireless sensor networks to exchange data between devices. This Thesis solely focuses on the IPv6 support of the network stack.

The existing communication stack seems to be specifically tailored to IEEE 802.15.4 as link-layer technology and causes problems if other link-layers (e.g., BLE) should be supported. One problem is that the function for sending packets in the radio layer only accepts packets with a maximum length of 255 bytes, which is sufficient for IEEE 802.15.4 but may not be for other link layer technologies, and the sending function is expected to block until the packet was sent. This blocking behavior can easily be implemented for IEEE 802.15.4 radios but causes problems with wireless technologies like BLE where the actual sending of packets is done by a separate processor and the radio layer simply adds packets to the transmission queue.

Listing 2.1: Example of a `project-conf.h` file that sets CSMA as the MAC layer and ContikiMAC as RDC layer

```
/*---------------------------------------------*/
#ifndef PROJECT_CONF_H_
#define PROJECT_CONF_H_
/*---------------------------------------------*/
/* network stack settings */
#define NETSTACK_CONF_MAC       csma_driver
#define NETSTACK_CONF_RDC       contikimac_driver
/*---------------------------------------------*/
#endif /* PROJECT_CONF_H_ */
/*---------------------------------------------*/
```

The Contiki OS provides different implementations for each of the communication stack layers. The different implementations can be used to customize the behavior of the communication stack individually for each Contiki user application. The configuration of the layers is done in an application-specific header file (`project-conf.h`) that is typically located in the root directory of the Contiki application. The example `project-conf.h` header file shown in Listing 2.1 selects CSMA (`csma_driver`) as the MAC layer and ContikiMAC (`contikimac_driver`) as the RDC layer to be used.

2.4 Hardware

This section lists the hardware used in this Thesis:

2.4.1 Texas Instruments SensorTag

Figure 2.12: TI SensorTag (`http://www.ti.com/ww/en/wireless_connectivity/sensortag2015/images/sensorTag-main-visual.png`).

This SoC (Figure 2.12) is based on the Texas Instruments (TI) CC2650, an ultra-low power MCU which supports Bluetooth LE and IEEE 802.15.4 wireless technologies. The CC2650 consists of a radio core and an application core.

The radio core is an ARM Cortex-M0 processor, which implements the lower functionality of BLE and IEEE 802.15.4. This radio core performs BLE and IEEE 802.15.4 packet reception and transmission and can run autonomously from the rest of the system. The application core, an ARM Cortex-M3 processor, runs the user application. This processor operates at 48 MHz, provides 128 KB of flash memory and 20 KB of RAM. A single CR2032 coincell battery provides the power supply of the whole SoC. [28]

The SensorTag is one of the three primary hardware platforms supported by Contiki [43]. Because the hardware supports both Bluetooth Low Energy and IEEE 802.15.4, it is provides the opportunity to compare both wireless protocols on the same hardware.

Figure 2.13: Nordic Semiconductor nRF52 DK (`http://www.nordicsemi.com/eng/Products/Bluetooth-Low-Energy2/nRF52-DK`).

2.4.2 Nordic Semiconductor nRF52

The nRF52 from Nordic Semiconductor (Figure 2.13) is an ultra-low power SoC that supports BLE and the ANT protocol for wireless data exchange. The SoC comes with a so called SoftDevice that provides applications with a full BLE communication stack and other hardware abstractions. The SoftDevice is freely available online but its source code is proprietary.

The applications run on an ARM Cortex-M4 processor that operates at 64 MHz, has 64 kB of RAM and 512 kB of flash memory. Power supply of the SoC can either be a single coin cell battery (CR2032) or an external power supply with a voltage range from 1.7 V to 3.6 V. [36]

Currently the nRF52 is officially supported by the Contiki OS [43] and source code supporting the nRF52 hardware platform is already merged into the official Contiki GitHub repository [8].

2.4.3 Raspberry Pi

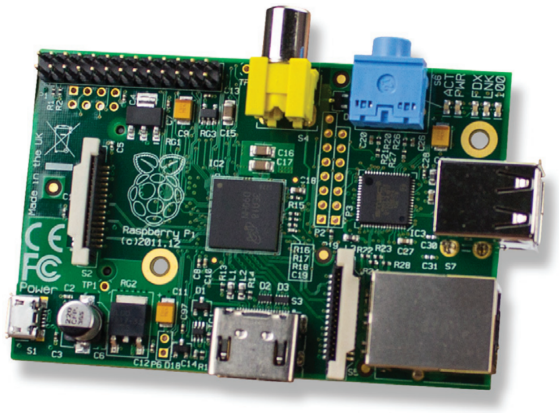

Figure 2.14: Raspberry Pi 1 Model B [13].

The Raspberry Pi 1 Model B computer (Figure 2.14) is used as a border router in

this Thesis. This Raspberry Pi version contains an ARM11 application processor that operates at 700 MHz and provides 512 MB of RAM. The Pi is powered using its Micro USB socket by an external power supply with 5 VDC [13].

For this Thesis the Raspberry Pi uses the Raspbian 8[1] as its Operating System. This OS provides all necessary libraries to configure the Raspberry Pi as border router.

Since this version of the Raspberry Pi does not provide BLE support a BLE to USB dongle, the LogiLink BT0015[2] was used to enable the Raspberry Pi to communicate with BLE devices.

[1] https://www.raspberrypi.org/downloads/raspbian/
[2] http://www.logilink.eu/showproduct/BT0015.htm

Chapter 3

Related work

This chapter discusses the related work to this Thesis. Section 3.1 lists different BLE stack implementations for constrained devices. Other studies closely related to this Thesis are summarized in Section 3.2.

3.1 BLE stacks

This section summarizes the most relevant implementations of BLE communication stack for constrained devices available to data. Out of these, only one communication stack (described in Section 3.1.1) provides support for IPv6 over BLE. Apache Mynewt described in Section 3.1.2 is currently under development and is expected to be the first fully open source BLE stack that provides IPv6 support, although no expected release date for the IPv6 support is available.

Table 3.1: Overview of existing BLE stack implementations for constrained devices.

	Open Source	IPv6	Contiki compatible	Portability
Nordic Semiconductor - BLE stack	NO	YES	NO	NO
Texas Instruments - BLE stack	NO	NO	NO	NO
Apache Mynewt - NimBLE	YES	NO (planned)	NO	YES
Bluekitchen - BLE stack	YES	NO	YES	NO
Cypress Semiconductor - WICED Smart	NO	NO	NO	NO
Blessed - BLE stack	YES	NO	NO	YES

Table 3.1 gives an overview of the existing BLE stack implementations for constrained devices. Each of the implementations is evaluated according to the requirements of this Thesis:

- Is the implementation of the BLE host fully open source?
- Does the stack support IPv6 over BLE?
- Is the implementation compatible to the architecture of the Contiki OS and its network stack?
- Could the implementation easily be ported to other hardware platforms, especially to the TI SensorTag?

3.1.1 Nordic Semiconductor - BLE stack

An implementation of IPv6 over BLE according to the RFC 7668 standard [34] was recently included into the official Contiki codebase [8]. This project implements IPv6 communication over a BLE link layer on the nRF52 DK platform of Nordic Semiconductor. The code uses a proprietary and closed source BLE communication stack that is provided by Nordic Semiconductor and is only available for certain Nordic Semiconductor hardware platforms. Thus the code providing the IPv6 over BLE functionality for the nRF52 DK platform cannot be easily ported to other hardware platforms, which are not supported by the Nordic Semiconductor BLE stack.

Another drawback of the used BLE stack is that this implementation is not fully compatible with the architecture of the Contiki OS network stack. The radio and duty cycling functionality, which are normally implemented in the respective layers of the Contiki network stack, are handled in the proprietary stack.

3.1.2 Apache Mynewt - NimBLE

Apache NimBLE is part of Apache Mynewt, a modular OS for IoT devices, and claims to be the world's first fully open source BLE stack. NimBLE is compliant to the Bluetooth Specification version 4.2 [5] and also supports features from older Bluetooth specifications such as multiple simultaneous roles of a BLE device and simultaneous advertising and scanning [20].

The currently supported platforms of Apache Mynewt are nRF51 DK and nRF52 DK from Nordic Semiconductor, BMD-300-EVAL-ES from Rigado, STM32F3DISCOVERY from ST Micro, STM32-E407 from Olimex and the Arduino Zero, Zero Pro and M0 Pro [19]. Because NimBLE is part of the Apache Mynewt OS and due to the architectural differences between Apache Mynewt and the Contiki OS, NimBLE is not compliant with the network stack architecture of Contiki.

The project is currently under development and as of the current version 0.9.0 of Apache Mynewt neither support for IPv6 over BLE nor any expected release date of IPv6 over BLE support is available [20].

3.1.3 Texas Instruments - BLE stack

Texas Instruments provides a proprietary BLE communication stack for its CC2640 and CC2650 hardware platforms. This BLE stack complies to the Bluetooth Specification 4.1 [3] and is part of the TI-RTOS, a real time operating system for Texas Instruments devices. In addition to the full support of Bluetooth Specification 4.1 the BLE stack implements

simultaneous master and slave operations. However, although the stack provides full BLE support, IPv6 over BLE communication is not supported [26].

Since the BLE stack is proprietary, it cannot easily be ported to hardware other than the supported Texas Instruments platforms. Due to the fact that the stack is not open source and it is part of the TI-RTOS, the stack is also not compatible with the Contiki network stack architecture.

3.1.4 Bluekitchen - BLE stack

The communication stack from BlueKitchen is an open source stack which not only provides BLE, but also classic Bluetooth functionality according the Bluetooth Specification 4.2 [5]. While it claims to have a small memory footprint (40kB flash and 4kB of RAM), it includes code for classic Bluetooth that is not needed in constrained devices [21].

This stack can easily be ported and could also be merged into the architecture of the network stack of Contiki. Unfortunately, the BLE stack does not provide support for IPv6 over BLE communication.

3.1.5 Cypress Semiconductor - WICED Smart

Cypress Semiconductor provide several SoC solutions with BLE wireless connectivity according the Bluetooth Specification 4.2 [5]. The SoCs come with corresponding WICED Software Development Kits that enable the creation of applications with BLE connectivity.

The code of the communication stack is available online without charge but users need to register to download the code base. Although WICED supports several BLE applications (e.g., proximity, thermometer) it seems that IPv6 over BLE is currently not supported [14].

3.1.6 Blessed - BLE stack

Blessed is a BLE software stack for embedded devices. This stack only supports BLE advertising and passive scanning, a BLE connection between two devices cannot be established. The project page shows that features like active scanning and BLE connections were planned to be added, but the last commit to the projects source code was done over 2 years ago and I assume it has been discontinued [10].

3.2 Studies

Dementyev et al. [16] compared the energy consumption of BLE, ANT and IEEE 802.15.4 radios used in constrained devices. In their setup, the authors periodically send small data packets to another radio device (e.g., health monitor applications) and measured the energy consumption of the three different radio devices. The authors came to the conclusion that BLE has the lowest power consumption compared to ANT and IEEE 802.15.4, at least in their experimental setup (8 bytes of data every 5 - 120 seconds).

Another comparison of the energy consumption of BLE and IEEE 802.15.4 was carried out by Siekkinen et al. [40] and shows that BLE has an extremely small energy consumption and a very attractive energy per bit ratio. Their results suggest that the energy to

bit ratio of BLE gets better the more bytes are transmitted during a BLE connection event. Additionally to the energy consumption evaluation, Siekkinen et al. calculated the overhead of IPv6 over BLE according to their energy measurements and concluded that the overhead of 6LoWPAN headers for IPv6 over BLE is the same as for IPv6 over IEEE 802.15.4 (e.g., 2 bytes 6LoWPAN header for link-local communication).

Narendra et al. [33] compared BLE to IEEE 802.15.4 regarding their link layer performance. They evaluated the link layer latency, the duty cycle of the radios, the maximum data rate, and the packet reception rate of different configurations of both link layer technologies. They conclude that both BLE and IEEE 802.15.4 have their own properties and may be suitable for different applications and advocate IPv6 as the means to realize interoperability between different link layer technologies.

The experiment performed by Chawathaworncharoen et al. [12] shows the feasibility of 6LoWPAN on BLE using commodity hardware (Raspberry Pi as nodes and laptop PC as border router). Their measurements show that the energy expenditure of IPv6 over WiFi is tenfold the energy expenditure of IPv6 over BLE in their experimental setup.

Chapter 4

IPv6 over BLE

This chapter presents the IPv6 over BLE communication stack for the Contiki OS that I implemented as part of this thesis work. Section 4.1 describes in detail the features and limitations of the implemented stack. Section 4.2 discusses the design decisions made during the development of the presented stack, so that the Contiki OS is able to use BLE as an alternative link layer technology to IEEE 802.15.4. Lastly, Section 4.3 summarizes the implementation details of the communication stack on the TI SensorTag and provides a guide for porting the created stack to other hardware platforms.

4.1 Features

This section describes in detail the features and limitations of the IPv6 over BLE communication stack implemented during my Thesis.

4.1.1 Open source implementation of the BLE host

All code created during this Thesis' work is fully open source under the 3-clause BSD license and available online[1]. Since no BLE stack for the Contiki OS is available at date, this implementation may be merged into the official Contiki repository. The availability of an open source BLE host may lead to additional features added to the existing implementation. Additionally, the stack can be easily ported to other BLE hardware platforms, as described in Section 4.3.3.

4.1.2 Interoperable communication stack compliant to RFC 7668

The IPv6 over BLE communication stack contains all necessary BLE features to setup the IPv6 over BLE connection and exchange IPv6 packets over the BLE connection. The stack is compliant to the RFC 7668 standard [34] and hence fully interoperable with compliant border routers.

Unfortunately, the Contiki OS version available at date is missing a feature that is needed to be fully compliant to the RFC 7668. Although it implements the IPv6 Neighbor Discovery mechanisms for IEEE 802.15.4 networks as specified by the RFC 4944

[1] `https://github.com/spoerk/contiki`

standard [32], the Contiki OS lacks support for IPv6 Neighbor Discovery Optimizations for 6LoWPANs as defined in the RFC 6775 [46]. Because the newer RFC 6775 is currently not supported by the Contiki OS, the packets exchanged during Neighbor Discovery do not contain the Address Registration Option as specified by the RFC 7668. Nevertheless, the implementation presented in this Thesis is fully interoperable with the existing Linux implementation of IPv6 over BLE border routers.

4.1.3 Contiki compatible communication stack

The extended communication stack is fully compatible with the architecture of the Contiki OS and its existing communication stack. Each of the stack layers implement the interfaces defined by the Contiki communication stack and could therefore be easily exchanged with other stack layer implementations that implement the same interface. Contiki application developers can easily change the stack layer implementations by changing the defined values in the `project-conf.h` file and hence switch from BLE to IEEE 802.15.4 as used link layer. A brief description of the configuration of the Contiki communication stack using the `project-conf.h` file can be found in Section 2.3.2.

4.2 Design

This section presents the design of the IPv6 over BLE communication stack of the Contiki OS. Section 4.2.1 summarizes the basic primitives that a node device needs to implement to exchange IPv6 packets over a BLE link. Section 4.2.2 discussed the overall design of the extended communication stack and the design of each of the stack layers. Section 4.2.3 describes the communication setup between node and border router in detail. Lastly, Section 4.2.4 discusses the design challenges faced during the development of the IPv6 over BLE communication stack for the Contiki OS.

4.2.1 Basic primitives

The three basic primitives of a IPv6 over BLE node device are:

- broadcasting IPv6 node behavior and allowing border routers to establish an IPv6 connection to the node;
- receiving BLE data packets (either control PDU or data PDU) during BLE connection events as a BLE slave device;
- transmitting BLE data packets (either control PDU or data PDU) during BLE connection events as a BLE slave device.

4.2.2 Communication stack

As stated in Section 4.1, one constraint of the IPv6 over BLE communication stack is that it should be fully compatible with the architecture of the existing Contiki communication stack. Figure 4.1 shows the communication stack with support for BLE as a link layer that I designed. This stack is compatible with the existing Contiki OS architecture and uses

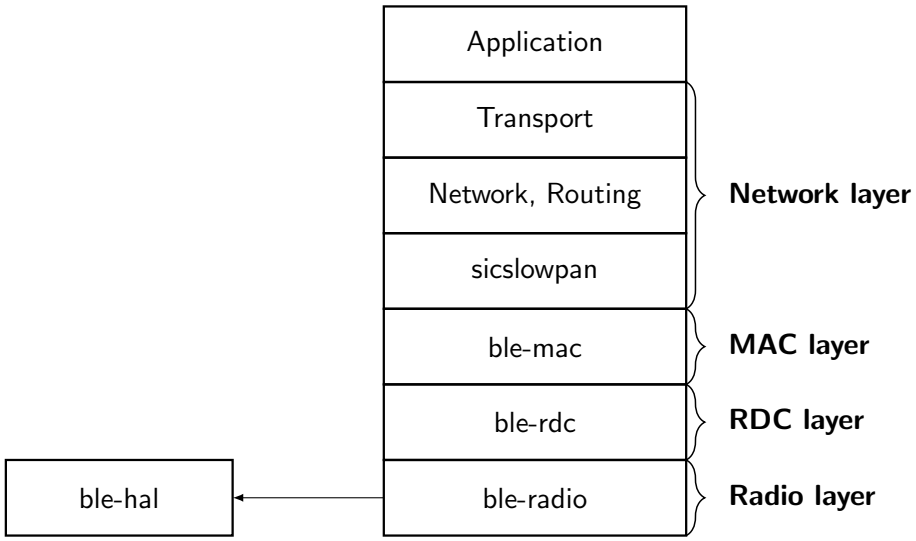

Figure 4.1: Contiki communication stack with IPv6 over BLE support

the same layers of Contiki's communication stack: radio layer, RDC layer, MAC layer, and network layer.

To switch from the classic IPv6 over IEEE 802.15.4 to the implemented IPv6 over BLE a user simply needs to change the used communication stack layers and communication stack parameters in either the Contiki OS configuration file of the used hardware platform (`contiki-conf.h`) or directly in the application-specific configuration file (`project-conf.h`).

One of the main changes in the communication stack for IPv6 over BLE is the introduction of the BLE-HAL, a Hardware Abstraction Layer (HAL) that is part of the BLE radio layer and that hides to the developer the implementation details of the various supported BLE hardware. Every hardware platform that is supported by the IPv6 over BLE stack needs indeed to provide a hardware-specific implementation of the BLE-HAL. By hiding the hardware-specific BLE code in the BLE-HAL implementations, the BLE radio layer is hardware independent and stays the same on all supported hardware platforms.

BLE-HAL

The BLE-HAL is a hardware abstraction layer that provides standardized functions to the radio layer in order to interact with the specific BLE radio hardware of the used hardware platform. This HAL is specific to IPv6 over BLE and is not a separate layer in the communication stack but is rather used by the radio layer implementation to communicate with the BLE hardware of the system. All provided functions mimic HCI functions specified in the Bluetooth Specification [3]. This should ease implementing a BLE-HAL on hardware platforms that have a BLE radio supporting HCI commands (e.g., the Nordic Semiconductor NRF52 transceiver described in Section 2.4.2). Other BLE hardware platforms without HCI support (e.g., the TI SensorTag described in Section 2.4.1) can still implement the HAL as shown in this Thesis.

The minimal functions provided by BLE-HAL implementations are (see `ble-hal.h`):

- `reset`
  Resets the link-layer of the BLE radio. After a reset, the link-layer is in the Standby state.
- `read_bd_addr`
  Reads the public device address of the BLE radio.
- `set_adv_data`
  Sets the advertising data used during BLE advertising. The advertising data length is limited to 31 bytes.
- `set_scan_resp_data`
  Sets the scan response data used during BLE advertising. The scan response data length is limited to 31 bytes.
- `set_adv_param`
  Sets the parameters used during BLE advertising. Such parameters include advertising interval and used advertising channels.
- `set_adv_enable`
  Enables or disables BLE advertising on the BLE radio. The used advertising parameters, advertising data and scan response data need to be set using the respective functions before enabling advertising.
- `disconnect`
  Disconnects a BLE connection that was previously established by the peer device.
- `send`
  Adds the specified data to the BLE transmission queue. The data is only added if the BLE radio is in the Connected state (either as master or slave device) and the specified data is then transmitted during the BLE connection events.

  The data length is limited to 255 bytes. Fragmentation of the specified data into BLE data packets with a maximum length of 27 bytes (as seen in Figure 4.2) is handled in implementations of the BLE-HAL.

Besides providing these functions for interacting with the BLE radio, the BLE-HAL copies the payload of BLE data packets that are received during connection events into the packet buffer of the Contiki communication stack and notifies the RDC layer that BLE data is available by calling `NETSTACK_RDC.input()`.

Radio layer

Typically the radio layer of Contiki is responsible for transmitting and receiving data packets over wireless technologies (e.g., IEEE 802.15.4). In the case of BLE as link layer, the transmission and reception of BLE data packets is the main task of the radio layer. The latter provides functions to start BLE advertisement and exchange BLE data packets during connection events.

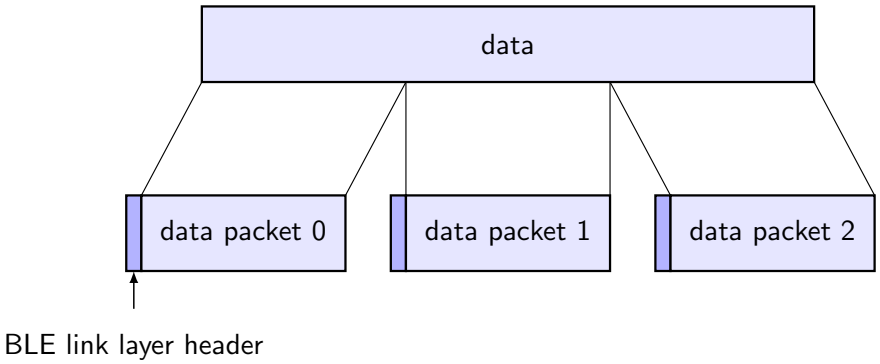

Figure 4.2: Fragmentation of data into several BLE data packets

As stated in Section 4.2.2 the radio layer uses a hardware specific implementation of the BLE-HAL to communicate with the BLE radio. Since all the hardware specifics are handled in the BLE-HAL, the radio layer for BLE is hardware independent and may be used for different BLE hardware platforms.

The `radio.h` file defines the interface of each implementation of a radio layer in the Contiki communication stack. Although this interface specifies 14 different radio layer functions, the radio layer for BLE devices needs to implement at least the radio layer functions:

- `init`
  Calls `init` of the BLE-HAL implementation to reset the BLE radio hardware.

- `send`
  This function is used by the upper communication stack layers to transmit data over BLE. The data length is limited to 255 bytes because of the architecture of the Contiki communication stack (payload length is an `unsigned short` in `radio.h`). When called, this function uses the `send()` primitive of the BLE-HAL to add the provided payload to the transmission queue of the BLE radio.

  Unlike the implementations of this function for other wireless technologies like IEEE 802.15.4 radios, this function does not block until the radio has transmitted the payload. Since the payload data is only appended to the transmission queue of the BLE radio and the radio then transmits the data deferred during connection events, this function immediately returns with `RADIO_TX_OK` after the payload is successfully appended. In case the payload could not be appended, this function returns with `RADIO_TX_ERR`.

- `on`
  This empty function always returns `1` and is only available for compatibility to the communication stack.

- `off`
  Calls `disconnect` of the BLE-HAL to disconnect any BLE connection.

- `get_value`
  Can be used to read various parameters and constants of the BLE hardware like buffer size and minimum and maximum data channel.
- `set_value`
  Can be used to set specific parameters for the BLE radio like advertisement interval and used advertisement channels. This function is also used to enable or disable BLE advertisement.
- `get_object`
  Can be used to read the BLE device address. The device address is read using `read_bd_addr` of the BLE-HAL.
- `set_object`
  Can be used to set the advertisement payload or the scan response payload. Therefore the BLE-HAL functions `set_adv_data` and `set_scan_resp_data` are used respectively.

RDC layer

Existing RDC layer implementations (e.g., `ContikiMAC` or `CXMAC`) implement duty cycling of the used radio layer. Radio duty cycling of BLE communication is already implemented in the link layer of BLE and performed by the BLE radio core as specified by the Bluetooth specification [3]. By using existing RDC layer implementations such as `nullMAC`, which do not perform any duty cycling of the radio, the application processor does not interfere with the existing BLE duty cycling of the BLE radio core.

In contrast to the existing RDC layer implementations that duty cycle the radio by calling `NETSTACK_RADIO.on()` and `NETSTACK_RADIO.off()`, duty cycling of the BLE radio should not frequently switch the radio on and off, because the basic duty cycling is already implemented in the BLE-HAL. Instead, RDC implementations for BLE radios may indirectly change the radio duty cycle by **changing the connection interval** and **slave latency** of the active BLE connection. These two connection parameters could be adaptively changed depending on the data to be sent or received over the BLE connection. Another approach could be to completely shut down the BLE radio core for longer periods of time (e.g., several minutes or hours) and only start the BLE connection for short periods to exchange data.

RDC layer implementations used for IPv6 over BLE have to implement at least the following functions defined by `rdc.h`:

- `init`
  Initializes the RDC layer implementation.
- `send`
  This function is used by the MAC layer to transmit data. `NETSTACK_RADIO.send()` is called to send data over BLE connections.
- `input`
  Called by the radio layer if any new data was received. This function calls `NETSTACK_MAC.input()` to notify the upper stack layers that valid data is available.

- `on`
  Depending on the implementation this function may enable the radio layer using `NETSTACK_RADIO.on()`.

- `off`
  Depending on the implementation this function may disable the radio layer using `NETSTACK_RADIO.off()`.

In future work, this layer could use BLE advertising primitives (advertising and scanning) to exchange IPv6 packets. These primitives are currently only used for communication setup between node and border router. Using BLE advertising and scanning directly could result in a very energy efficient way to exchange IPv6 packets between BLE devices.

MAC layer

For IPv6 over BLE the main purpose of the MAC layer is to implement the L2CAP functionality of BLE.

During initialization this layer configures and starts the BLE advertisement to indicate the IPv6 over BLE support to nearby border routers as specified by [4]. After a link layer connection has been created by the BLE radio, the MAC layer handles the L2CAP channel setup by parsing and responding to the L2CAP connection request of the border router and hence creating an LE Connection Oriented L2CAP Channel with LE Credit Based Flow Control Mode [4].

Although the fragmentation of a single IPv6 packet into smaller radio packets is usually done by the network layer in the 6LoWPAN implementation, the RFC 7668 standard specifies that L2CAP performs fragmentation and reassembly of data [34].

MAC layer implementations used for IPv6 over BLE have to implement at least the following functions defined by `mac.h`:

- `init`
  Configures the BLE advertisement parameter, advertisement and scan response data according to [4] and starts BLE advertisement by calling `NETSTACK_RADIO.set_value()` and `NETSTACK_RADIO.set_object()` with the according parameters.

- `send`
  Called by the Network layer to transmit an IPv6 packet. The maximum length of the IPv6 packet is 1280 bytes (the minimum MTU specified by [4, 34]). As stated above, this layer handles fragmentation of a single IPv6 packet into L2CAP packets as shown in Figure 4.3. The L2CAP packets are then sent by calling `NETSTACK_RDC.send()`.

- `input`
  Called by the RDC layer if any new data was received.

  The reassembly of L2CAP packets into IPv6 packets is implemented in this function. If a whole IPv6 packet was received the Network layer is notified by calling `NETSTACK_LLSEC.input()`.

- `on`
  Depending on the implementation this function may enable the RDC layer using `NETSTACK_RDC.on()`.

- `off`
  Depending on the implementation this function may disable the RDC layer using `NETSTACK_RDC.off()`.

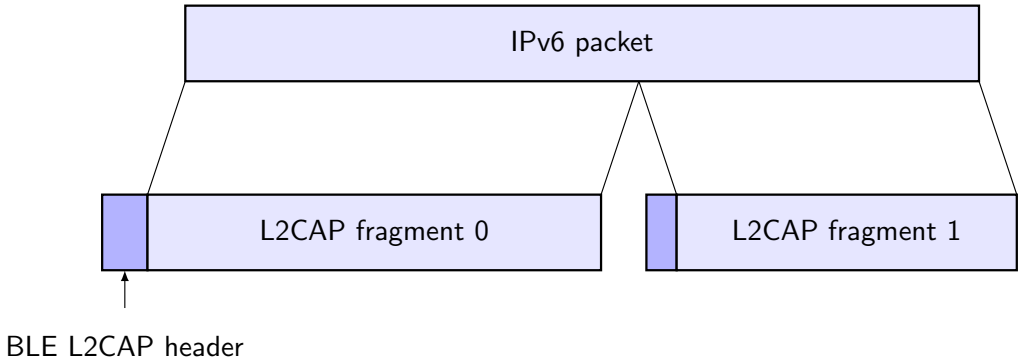

Figure 4.3: Fragmentation of a single IPv6 packet into L2CAP fragments

Network layer

The existing Network layer of Contiki does not need to be changed to support IPv6 over BLE. The only necessary modifications are changes in the parameters of the Network layer to disable fragmentation and reassembly in the Network layer since this is done in L2CAP as specified in RFC 7668 [34].

4.2.3 Communication setup

The communication setup between a node device and the border router is a four step process that is shown in Figure 4.4. At first the node device sends out BLE advertisement packets to all BLE devices in its surrounding and waits for the border router to establish a BLE link-layer connection (see Section 4.2.3). Secondly the two devices create a L2CAP connection on top of the established BLE link-layer connection (see Section 4.2.3) and thereby establish the IPv6 connection that uses the L2CAP channel to exchange data. Thirdly the node sends router solicitation messages to the border router and receives router advertisement messages that carry necessary subnet information like the shared IPv6 prefix (see Section 4.2.3). At last all non-link-local IPv6 addresses of the node are registered at the router using neighbor solicitation messages that are acknowledged by neighbor advertisements (see Section 4.2.3).

By adhering to this communication setup routine specified by the BLE specification of the Internet Protocol Support Profile [4] and the RFC 7668 standard [34], my implemented IPv6 over BLE communication stack does **not** require any change in border routers that support the RFC 7668 standard.

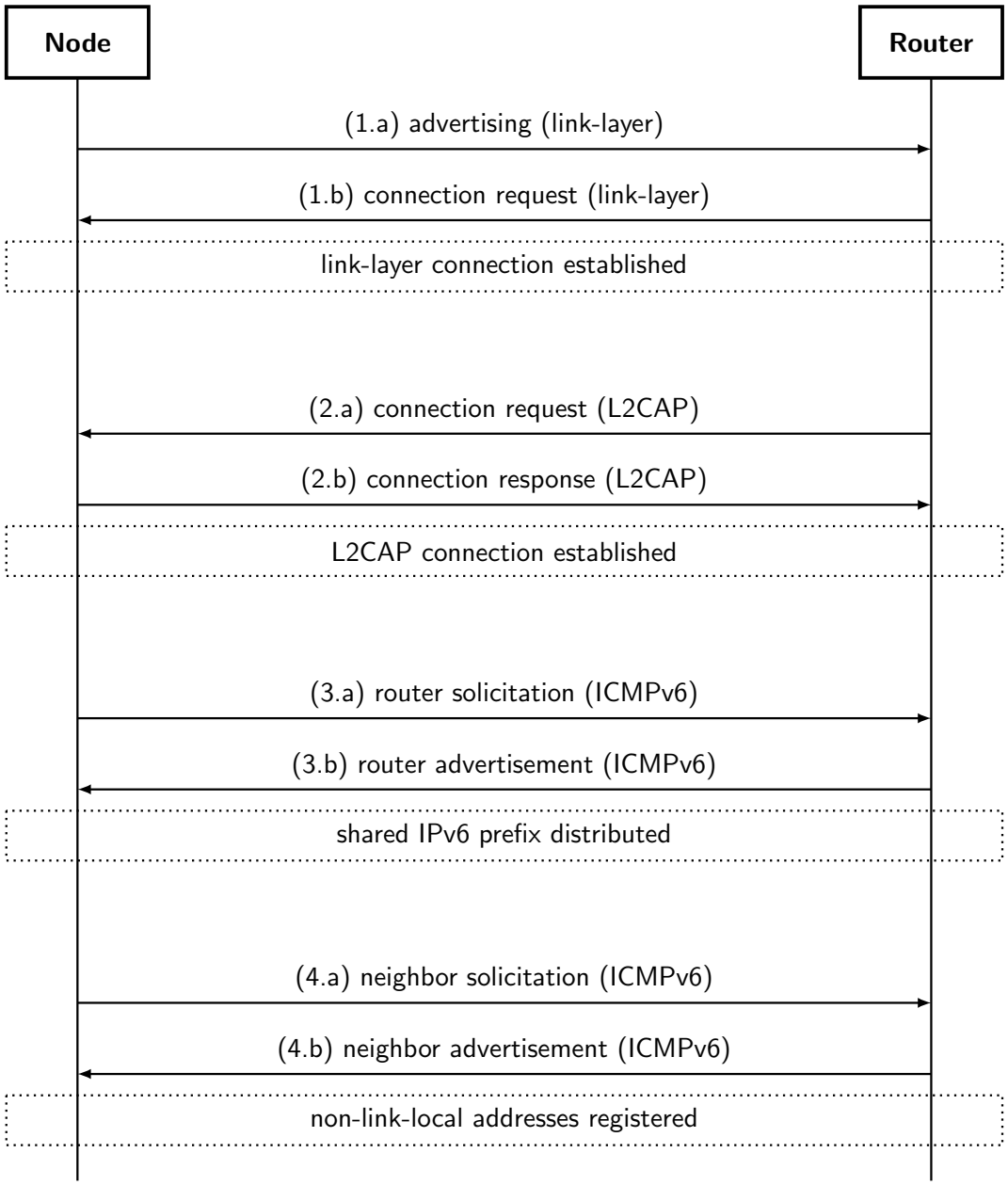

Figure 4.4: The 4 steps of a connection setup between a node and a border router; first a BLE link-layer data connection is established (steps 1.a and 1.b), second a BLE L2CAP connection is created (steps 2.a and 2.b), then the node uses ICMPv6 messages to find the router and the shared IPv6 prefix (steps 3.a and 3.b) and register its non-link-local addresses at the border router (steps 4.a and 4.b).

BLE link-layer connection

Nodes that do not have a connection to any border router send out BLE connectable advertisement packets to allow routers to discover it [4] (step 1.a in Figure 4.4). Devices

Table 4.1: Advertisement data fields that need to be defined by a IPv6 over BLE node to enable connection setup [6].

Flags	defines if the BLE device is general or limited discoverable and if the device supports BR or EDR Bluetooth modes
TX Power Level	indicates the transmission power level of sent packet
Local Name	shortened or complete name of the BLE device
Service UUIDs	a list of the services provided by the BLE device; the IPSS needs to be in this list
Slave Connection Interval Range	the preferred connection interval range of the BLE device

requesting to be connected to a IPv6 over BLE border router need to implement a GATT server role and shall instantiate the Internet Protocol Support Service (IPSS) as a primary GATT service.

According to the GATT specification in [3] and [6] and the additional requirement for IPv6 over BLE node to instantiate the IPSS service defined by [4], the advertising node has to at least define the following values: Flags, TX Power Level, Local Name, Service UUIDs and Slave Connection Interval Range, as summarized in Table 4.1.

When a border router discovers an advertising node that wants to connect to the Internet, the router checks for duplicate link-local addresses and sends a connection request to the advertising node (step 1.b in Figure 4.4). With this connection request the router initiates the BLE link-layer data connection and sends the necessary connection parameters to the node. After the node has successfully received the connection request, the link-layer connection is established and all exchanged data between the node and the router is sent on one of the BLE data channels using BLE connection events [3].

BLE L2CAP connection

After the link-layer connection was successfully established, the border router initiates a L2CAP connection between the router and the node (step 2.a in Figure 4.4). An L2CAP connection is needed because each IPv6 packet is embedded into one or more L2CAP packets and sent over the L2CAP connection. The successful creation of the channel is acknowledged by the node by sending an L2CAP connection response to the router (step 2.b in Figure 4.4). When the L2CAP channel is successfully created all subsequent communication between node and router is done using IPv6 packets sent over the established L2CAP connection.

The definition of the IPSS [4] states that the LE Connection Oriented Channel with LE Credit Based Flow Control is used for the L2CAP connection between the router and the node. The main differences between the LE Connection Oriented Channel with LE Credit Based Flow Control mode and the other modes of L2CAP are that the used mode allocates the L2CAP channel identifiers dynamically and that each device may limit the incoming L2CAP packets using credits as a flow control mechanism. During the L2CAP channel setup the node grants the border router an initial amount of credits and vice versa. These credits represent the number of L2CAP packets that the granting device accepts on the L2CAP channel. After an L2CAP packet was received the credits are decremented by

one and if the credit counter reaches 0 no more L2CAP packets are accepted. However either border router or node may grant additional credits using an L2CAP Flow Control Credit packet.

Router solicitation and advertisement

IPv6 over BLE uses neighbor discovery mechanisms as defined in [46]. The latter standard describes discovery approaches for several 6LoWPAN topologies including link-layer technologies supporting mesh networking like IEEE 802.15.4. Since BLE does not support mesh topologies, only the neighbor discovery approaches concerning star topologies are used in implementations according to RFC 7668 [34].

A node sends Router Solicitations (RSs) to the all-routers multicast address when its router list is empty, the default router is not reachable, or the lifetime of the router information is expired [46]. This means that after the L2CAP connection to the router is established and hence IPv6 communication is possible, the node sends a RS to the all-routers multicast address and waits for a Router Advertisement (RA) from the router.

When the router receives the RS, it responds with a RA message as defined in [42] and [46]. Such a RA carries the IPv6 prefix for the subnet and possibly contexts that is used in the IPv6 header compression of 6LoWPAN.

Neighbor solicitation and advertisement

Using the prefix information received with the RA from the router, the node creates its global address and registers this created address. A node registers the created global address at the router using a Neighbor Solicitation (NS) message with the address registration option set. Any other global address of the node may be added to the NS using another address registration option field [46].

The router checks the received global address for duplicates in its neighbor cache and acknowledges the successful address registration by sending a valid Neighbor Advertisement (NA) to the node [46].

4.2.4 Design challenges

The biggest challenge while designing the IPv6 over BLE communication stack was that the whole Contiki OS was designed around IEEE 802.15.4 as the link layer technology. This specifically caused a problem while designing the radio layer of BLE. The whole existing communication stack requires the `send()` function of the radio layer to block until the packet to be sent is successfully transmitted or the transmission was unsuccessful (e.g., due to collisions). In contrast to a IEEE 802.15.4 radio, that could try to transmit a packet any time, BLE only allows to exchange data during connection events, that are fixed in their start and length. This means that the `send()` function of the BLE radio layer cannot start transmitting data whenever `send()` is called. One solution for this problem would have been to store the BLE data to be sent in the transmission queue of BLE-HAL, perform a busy wait until the data was transmitted during a connection event, and then return from the `send()` function. Although this alternative solution would have led to a blocking `send()` function, it would have also consumed a significantly higher amount of energy than the existing design, where the application processor is idle during connection

events. To use this non-blocking radio layer, the upper layers of the connection stack need to be adapted. The MAC layer implementation needs to wait between sending two consecutive L2CAP fragments and needs to check if any buffer space is available in the BLE-HAL before adding another L2CAP fragment to the transmission buffer.

Another constraint encountered while creating the design of the new communication stack was the limited payload length of the `send()` function of the radio layer interface. The `send()` function supports a maximum payload length of 255 bytes as a parameter (the payload length is a variable of type `unsigned short`). This limited payload length is enough for IEEE 802.15.4 radios, since the maximum IEEE 802.15.4 payload length is 127 bytes [41]. Nevertheless, BLE supports payloads longer than 255 bytes, but because of this limitation the maximum length of the L2CAP fragments is limited to the 255 bytes. A possible solution would have been to change the radio interface of the Contiki communication stack, but since one of the requirements of the new communication stack is to be compatible to the existing Contiki OS, this solution was not chosen. Instead, I chose to use the existing radio interface and use a maximum L2CAP fragment size of 255 bytes in the BLE-MAC layer.

4.3 Implementation

This section describes the implementation of the IPv6 over BLE communication stack in Contiki that was created during this thesis. Section 4.3.1 discusses the implemented changes to the Contiki communication stack, especially the created stack layer implementations. Section 4.3.2 highlights the main challenges that I faced during implementing the IPv6 over BLE communication stack on the TI CC2650 SensorTag. Section 4.3.3 provides a guideline on how the existing implementation for the TI SensorTag can be ported to other hardware platforms with BLE support.

4.3.1 Communication stack

This section describes the changes made to the existing communication stack to support IPv6 communication over a BLE link.

BLE-HAL

The implementation of BLE-HAL that is used in this thesis can be found in the source file `contiki/cpu/cc26xx-cc13xx/rf-core/ble-hal/ble-hal-cc26xx.c`. This BLE-HAL implementation is specific to the Texas Instruments CC2650 and the communication between application and radio core is done using shared memory as specified by the Technical Reference Manual of the TI CC2650 [27]. Unfortunately, the radio core of the CC2650 does not implement the HCI.

The current implementation only provides the link-layer states: Standby, Advertising and Connection of BLE. The states Scanning and Initiating are not needed on IPv6 over BLE node devices and are therefore not implemented in the current version of the BLE-HAL. Figure 4.5 shows the three supported link-layer states and the valid transitions between these states. The states Scanning and Initiating are only shown to give an overview of all available link-layer states of BLE.

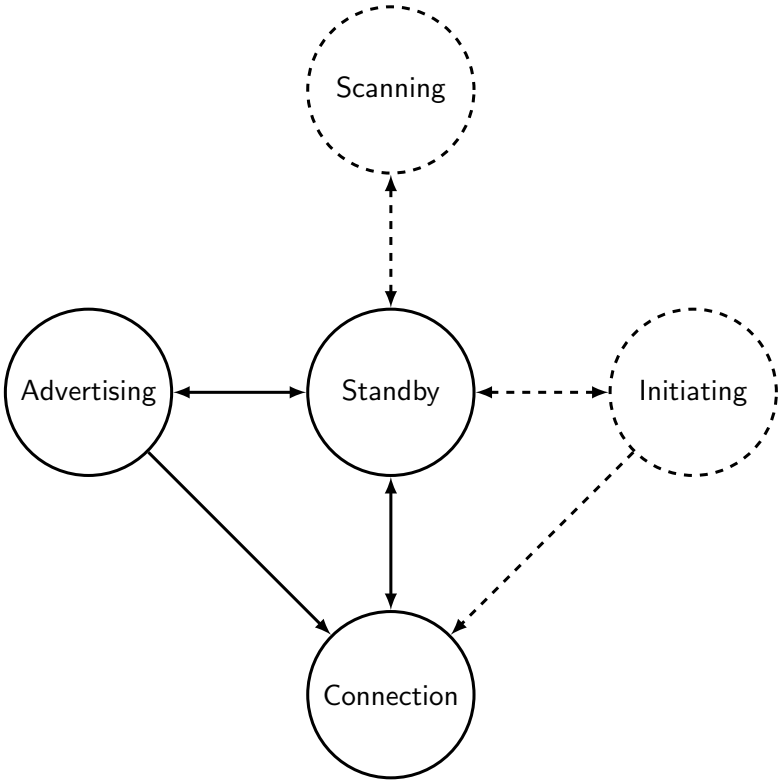

Figure 4.5: The state machine of the link-layer states of BLE. The states Standby, Advertising and Connection are currently implemented in the BLE-HAL of the TI CC2650 (adapted from [3]).

The main functionality of the BLE-HAL is implemented in the the `ble_hal_process`, a Contiki process that communicates with the BLE radio core of the TI CC2650. Unfortunately, the radio core of the TI CC2650 does not implement the HCI or even a full BLE link layer. Instead, the radio core only provides functions to start single BLE events (advertising and connection events), but the scheduling of consecutive advertising or connection events needs to be handled by the OS on the application core. Due to this limitation of the radio core, the `ble_hal_process` implements part of the link-layer of BLE and is responsible for scheduling advertising or connection events according to the Bluetooth Specification [3].

The `ble_hal_process` processes three different Contiki events to implement its behavior. Each of these three events is created and broadcasted (using `process_poll()`) when a specific interrupt from the BLE radio core was raised. The interrupt service routines for these interrupts are implemented in `contiki/cpu/cc26xx-cc13xx/rf-core/rf-core.c`. The events created and broadcasted are:

- `rf_core_data_rx_event`
  The radio core raises an interrupt when new BLE data was received during either an advertising or a connection event. When such an interrupt is detected by the application core, an `rf_core_data_rx_event` is created and posted.

- `rf_core_timer_event`
  The timer of the radio core can be used to raise an interrupt if a specific timer value has been reached. This timer interrupt can be used to time procedures that need to be synchronized with the radio core. When the application core detects such a radio timer interrupt, an `rf_core_timer_event` is created and posted.

  In the current implementation of the BLE-HAL, these `rf_core_timer_event`s are used to schedule advertising events.

- `rf_core_command_done_event`
  The radio core of the TI CC2650 raises an interrupt every time the last radio core command has been finished (a single connection event or a single advertisement event is closed). The application core listens for these interrupts and creates and posts `rf_core_command_done_event`s whenever such an interrupt is detected.

  The current BLE-HAL implementation uses these `rf_core_command_done_event`s to schedule connection events.

The BLE-HAL implementation uses two data packet queues to handle the communication over BLE advertising and data channels. The `rx_data_queue` is used to store BLE packets that are received during either advertising or connection events. It is a cyclic buffer providing space for 20 packets simultaneously. The `tx_data_queue` is used for sending data over BLE data channels. This queue is not a cyclic buffer, but buffer segments are dynamically appended to the `tx_data_queue`. The buffer space for the `tx_data_queue` is implemented using the Memory Block Allocator (`memb`) feature of the Contiki OS. In the current implementation of the BLE-HAL, the `tx_data_queue` holds a maximum of 60 BLE data packets (each has a data length of up to 27 bytes) simultaneously. BLE advertising and scan response data are not stored in the `tx_data_queue`, but set using `set_adv_data()` and `set_scan_resp_data()`, respectively.

When the BLE-HAL is in the Standby state both advertising and connection events are disabled. This state may always be entered by calling `disconnect` of the BLE-HAL.

In the Advertising state the process waits for either a `rf_core_data_rx_event` or a `rf_core_timer_event`. When a `rf_core_timer_event` is received the process configures the radio core to perform a single advertisement on each of the configured advertisement channels (see Figure 4.6). After the advertising event is closed, the process configures the radio core timer to raise an interrupt at the start of the next advertising event (after `advertising_interval`). A `rf_core_data_rx_event` is received when either a scan request or a connection request was received by the radio core during advertisement. When a valid connection request was received the process parses the connection parameters and switches into the Connection state.

The advertising parameter and data used are set via the functions `set_adv_data`, `set_scan_resp_data` and `set_adv_param`. These functions may be called in every state of the BLE-HAL (Standby, Advertising and Connection). The function `set_adv_enable` to enable BLE advertising only succeeds if the BLE-HAL is in the Standby state. Switching into Advertising from any other state but Standby is not supported, as shown in Figure 4.5.

In the Connection state the process waits for either a `rf_core_command_done_event` or a `rf_core_data_rx_event`. A `rf_core_command_done_event` indicates that the last

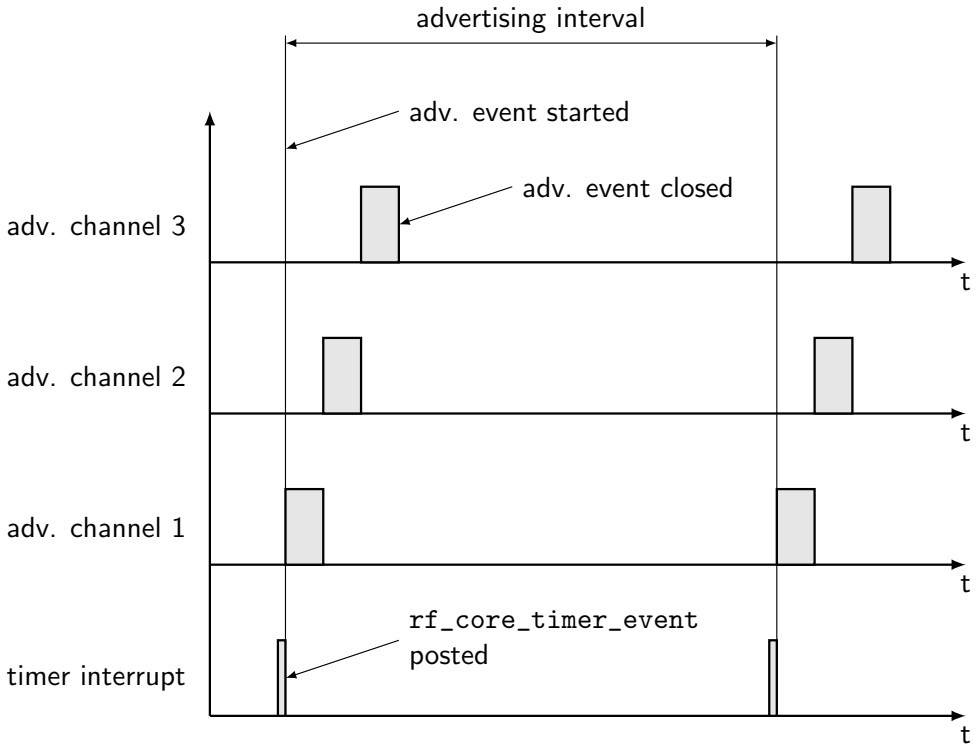

Figure 4.6: Timing diagram of two consecutive advertising events. In this diagram all three advertising channels are enabled to be used for advertising data.

connection event has been finished by the radio core. To configure and queue the next connection event, the BLE-HAL process removes successfully transmitted packets from the `tx_data_queue`, adds fresh packets to the `tx_data_queue`, calculates the start time and data channel of the next connection event, and queues the next connection event into the command queue of the radio core. The radio core starts the queued connection event autonomously at the configured start time, as shown in Figure 4.7, and handles the packet acknowledgment and flow control of all BLE packets.

When a `rf_core_data_rx_event` was received, the process parses the packets in the `rx_data_queue`. The received packets can either be a control packet (used to manage the BLE link-layer connection between two BLE controllers) or data packet (carrying data between two BLE controllers). If the received packet is a control data packet, the BLE-HAL process responds with the corresponding data. If one or more data packets were received, the payload of all packets is reassembled and copied into the packet buffer and the RDC layer is notified using `RDC.input()`. After the upper layers return, all finished packets in the `rx_data_queue` are freed so their space can be reused.

As already stated in Section 4.2.2 the `send` function of BLE-HAL implementations does not block until the given payload is transmitted. The function simply takes payload of up to 255 bytes, fragments the payload into BLE data packets of 27 bytes, adds those fragments to the `tx_data_queue`, and immediately returns.

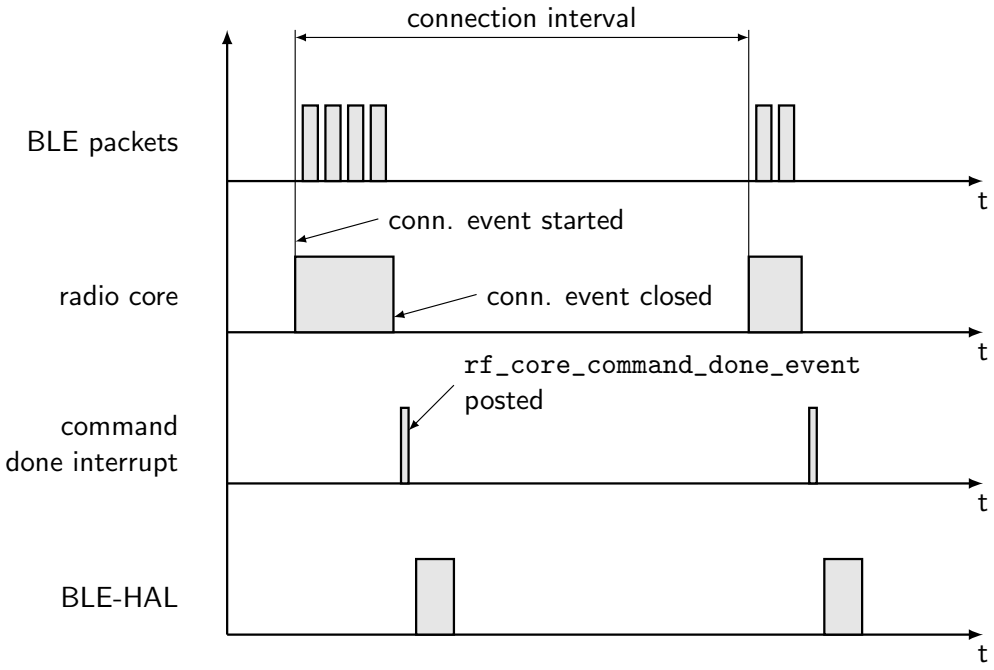

Figure 4.7: Timing diagram of two consecutive connection events showing the active time of the radio core and the BLE-HAL process in context with the exchanged BLE data packets and the radio core interrupt.

Radio layer

The file `contiki/cpu/cc26xx-cc13xx/rf-core/ble-mode.c` contains the implementation of the BLE radio layer. This implementation of the radio layer is generic and can be used for every BLE radio hardware that implements a BLE-HAL.

The `init` function of the radio layer calls the `init` function of the BLE-HAL and thereby initializes the BLE radio core of the CC2650. The function `on` is only implemented to be compliant with the Contiki network stack and is a dummy function that immediately returns with the value 1. Function `off` uses `disconnect` of the BLE-HAL implementation to terminate any ongoing BLE advertising or communication. The functions `get_value`, `set_value`, `get_object` and `set_object` are used to get and set paremeters (BLE device address, advertising data, advertising parameters) of the used BLE radio core. Any BLE specific additions to the existing `RADIO_PARAM` enumeration are defined in `ble-hal.h`.

Unlike the `send` functions of other radio layer implementations, this `send` function does not block until the data was successfully sent over BLE. If the payload was successfully queued into the `tx_data_queue` of the BLE-HAL implementation, `send` immediately returns with `RADIO_TX_OK`. Otherwise the `send` function returns with `RADIO_TX_ERR`.

RDC layer

The current IPv6 over BLE stack uses the existing `nullrdc-noframer` implementation of the RDC layer that is located in `contiki/core/net/mac/nullrdc-noframer.c`.

This implementation does not perform any duty cycling of the radio. All duty cycling

used in the current version of the IPv6 over BLE stack is done by the BLE radio core as specified in the Bluetooth Specification [3]. Future work may create new RDC layer implementations that change the duty cycle of the BLE radio and that are used instead of `nullrdc-noframer`.

MAC layer

The source file `contiki/cpu/cc26xx-cc13xx/net/ble-mac.c` contains the implementation of the MAC layer that was created for the IPv6 over BLE communication stack. As defined in Section 4.2.2, the MAC layer implementation for the BLE link-layer initiates the advertisement behavior of the node and is responsible for fragmentation and reassembly of large IPv6 packets by using the L2CAP functionality of BLE. The `ble-mac` uses two buffers for L2CAP fragmentation and reassembly: the `rx_buffer` for received data and the `tx_buffer` for outgoing data. Both of these buffers are able to store up to 1280 bytes.

Configuration of the BLE advertising behavior is done in the `init()` function. Using the functions `NETSTACK_RADIO.set_value()` and `NETSTACK_RADIO.set_object()`, the `init()` function sets the advertisement and scan response data and sets the advertising parameters (advertising interval, used advertisement channels). After the `init()` function has successfully been executed, the node device is indicating its IPv6 over BLE capabilities using BLE advertisement.

The `input()` function of the MAC layer handles incoming data packets and is called by the RDC layer when BLE data packets were received by the radio core. A received packet is either an L2CAP connection request, an L2CAP flow control credit or an L2CAP data packets. The L2CAP connection request is only received during the setup of the IPv6 over BLE connection between the node and the border router. When such a connection request was received, the `ble-mac` parses the connection request, creates the L2CAP connection-oriented flow channel, and responds with the corresponding L2CAP connection response. In the case of an L2CAP flow control credit, the packet is parsed and the L2CAP flow control credits are updated. When an L2CAP data packet is received, it is added to the `rx_buffer`. The Network layer is notified by calling `NETSTACK_LLSEC.input()`, when the `rx_buffer` contains a complete L2CAP message.

The `send()` function accepts IPv6 packets with a maximum length of 1280 bytes from the Network layer. `send()` copies the IPv6 packet to be sent into the `tx_buffer`, polls the `ble_mac_process` using `process_poll`, and immediately returns. In the case that the IPv6 packet has been successfully copied into the `tx_buffer`, the `sent_callback` is called with the result `MAC_TX_DEFERRED`. Otherwise, the `sent_callback` is called with the value `MAC_TX_ERR` if the length of the IPv6 packet is greater than 1280 bytes, or with `MAC_TX_COLLISION` if another IPv6 packet is currently being sent. In both of the latter cases the IPv6 packet does not get sent over the BLE connection.

The `ble_mac_process` is a Contiki process that handles the actual fragmentation and transmission of packets to be sent. When this process gets polled it checks if the `tx_buffer` contains data to be transmitted. The process creates and appends the L2CAP header in front of the data of the first L2CAP fragment and calls `NETSTACK_RDC.send()` to start transmission of the created fragment. If further L2CAP fragments need to be created and transmitted (the `tx_buffer` still has unsent data), the `ble_mac_process` waits using an `etimer` and sends the next fragment after the pause. The pause between two fragments

is needed to provide the other processes running (especially the `ble_hal_process`) with necessary CPU time. Figure 4.8 shows the procedure of fragmenting and sending a single IPv6 packet.

The current `ble-mac` implementation uses L2CAP fragments with a maximum length of 255 bytes and waits for approximately 15 ms between transmitting two consecutive L2CAP fragments.

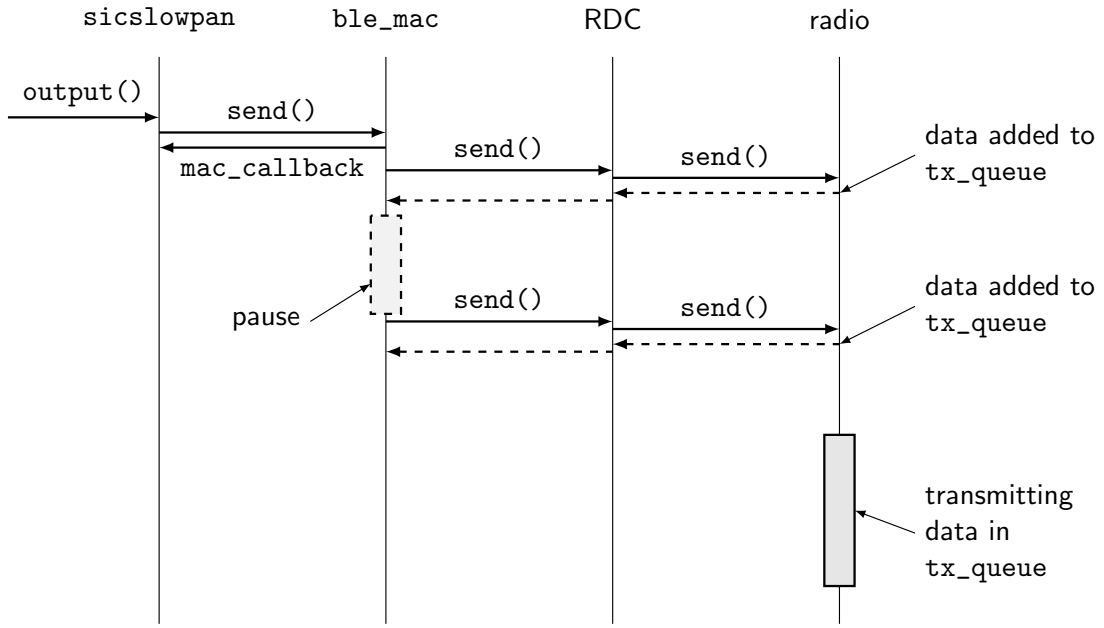

Figure 4.8: Diagram showing the process of sending a large IPv6 packet over BLE. The `ble_mac` implementation splits the IPv6 packet into two fragments.

Network layer

As defined in Section 4.2.2 the existing Network layer of the Contiki OS is used for IPv6 over BLE. Nevertheless, to support BLE as a link layer technology, some configuration parameters need to be changed to fit the BLE-specific needs. All of this parameters can be configured in either the `contiki-conf.h` file using `#define`, or in the Makefile of the used hardware platform.

The following parameters need to be changed compared to the classic IPv6 over IEEE 802.15.4 configuration to fully support the current implementation of the IPv6 over BLE stack (Table 4.2 summarizes parameter configuration):

- `SICSLOWPAN_CONF_MAC_MAX_PAYLOAD`
  This parameter defines the maximum payload length that the `sicslowpan` implementation passes to the MAC layer. If this parameter is not defined, the standard length of 125 bytes is choosen.

  Since the fragmentation of IPv6 over BLE is implemented in the MAC layer, this parameter needs to be set to 1280.

Table 4.2: Parameters of the Network layer that need to be changed for IPv6 over BLE. The column IEEE 802.15.4 shows the values of the standard communication stack while column BLE shows the configuration for IPv6 over BLE.

Parameter	**IEEE 802.15.4**	**BLE**
SICSLOWPAN_CONF_MAC_MAX_PAYLOAD	(default) 125	1280
SICSLOWPAN_CONF_COMPRESSION_THRESHOLD	63	0
SICSLOWPAN_CONF_FRAG	1	0
PACKETBUF_CONF_SIZE	(default) 128	1280
QUEUEBUF_CONF_NUM	8	1
CONTIKI_WITH_IPV6	1	1
CONTIKI_WITH_RPL	1	0

- SICSLOWPAN_CONF_COMPRESSION_THRESHOLD
  The `sicslowpan` implementation only compresses IPv6 packet that are larger than the threshold defined by this parameter.

  Unlike the IPv6 over IEEE 802.15.4 stack that only compresses IPv6 packets larger than 63 bytes, the IPv6 over BLE stack should compress all IPv6 packets and hence this parameter is set to 0.

- SICSLOWPAN_CONF_FRAG
  This parameter indicates if the fragmentation mechanism of the `sicslowpan` implementation is either enabled or disabled.

  Because the fragmentation of new stack is implemented in the MAC layer, this parameter needs to be set to 0 to disable the `sicslowpan` fragmentation.

- PACKETBUF_CONF_SIZE
  The maximum size of the packet buffer of the communication stack is defined by this parameter. The default value is 128 bytes and is used if this parameter is not defined.

  To support the MTU of IPv6, this parameter needs to be changed to 1280.

- QUEUEBUF_CONF_NUM
  The packet buffer of the communication stack is able to store several queued packets. This parameter defines the maximum number of queued packets supported.

  Due to memory constraints of the TI CC2650 and the necessary large value for PACKETBUF_CONF_SIZE, the current communication stack is only able to store a single packet in the packet buffer. Therefore this parameter is set to 1.

- CONTIKI_WITH_IPV6
  As stated in Section 2.3.2, the Contiki OS has two possible modes of the communication stack: IPv6 and RIME. Per default Contiki uses the RIME mode for the communication stack. To select the IPv6 mode of the stack, this parameter is set to 1.

- `CONTIKI_WITH_RPL`
  Contiki supports the RPL routing protocol when in IPv6 mode. This routing protocol is intended for multihop networks and is not needed for the IPv6 over BLE communication. To disable the RPL routing, this parameter is set to 0.

4.3.2 Implementation challenges

The biggest challenge implementing the IPv6 over BLE communication stack on the TI CC2650 SensorTag was that the SensorTag does not provide the standard HCI to communicate with the BLE link-layer. Moreover, the SensorTag does not even implement the whole link-layer functionality as specified by the Bluetooth Specification [3]. The radio core of the SensorTag only provides basic primitives to start single advertising or connection events. This means that instead of simply starting BLE advertising and letting the radio core schedule all future advertising events, the application core needs to handle the scheduling of each individual advertising event by itself. Other link-layer functionalities like parsing connection requests and scheduling connection events are also not supported by the radio core and need to be handled by the BLE-HAL implementation. This led to a complex BLE-HAL implementation for the TI SensorTag that implements BLE advertising and connection behavior. A radio core that implements all the link-layer functionality of BLE would have led to a much simpler and more energy efficient implementation of the BLE-HAL and hence to a more energy efficient and robust IPv6 over BLE communication stack.

Especially the scheduling of BLE connection events was a major difficulty in the BLE-HAL implementation. Although the start time of the connection event, the connection interval and the slave latency are known parameters, finding the right time to schedule the connection event on the BLE radio core is still a problem and may benefit from further optimizations. Even though the Bluetooth specifications [3] define some parameter to counteract inaccurate timing between master and slave, this so called “window widening” does not take the wakup time of the BLE radio core into account.

Another constraint during development was the limited memory of the SensorTag. Because the BLE-HAL and the BLE-MAC layer implementation need to fragment and reassemble data, both need large amount of buffer space to handle fragmentation. After memory optimizations in both of these layers, the current implementation of the IPv6 over BLE communication stack is able to run on a TI SensorTag and has free memory left to support complex Contiki applications. An evaluation of the memory consumption of the IPv6 over BLE stack can be found in Section 5.3.1.

4.3.3 Porting to other hardware platforms

This section provides a short guide on how to port the existing IPv6 over BLE communication stack implementation of the TI SensorTag to other Contiki hardware platforms that support BLE. The following components need to be adjusted to the new hardware:

BLE-HAL

The new hardware platform needs to provide a hardware specific implementation of the BLE-HAL. This implementation needs to adhere to the design presented in Section 4.2.2.

If the target hardware platform provides HCI support, this implementation may simply call the individual functions of the HCI needed. Other target hardware platforms that do not provide HCI support need to implement the necessary communication with the BLE radio core as stated in the hardware platforms documentation.

MAC layer

The implementation of the MAC layer may need to be adjusted in order to adhere to the memory constraints of the target hardware platform. Currently, the MAC layer implementation uses an L2CAP fragment size of 255 bytes. This fragment size may be decreased if the target platform has not enough memory to support such large fragments.

Additionally, the advertising parameters used during connection setup can be configured in the MAC layer implementation. Any changes to the advertising parameters, such as device name, additional service UUIDs, or the used transmission power may be changed in this layer.

Chapter 5

Evaluation

This chapter presents the evaluation of the implemented IPv6 over BLE communication stack. Section 5.1 shows the interoperability of this IPv6 over BLE communication stack with other IPv6 over BLE implementations. The network connectivity of the IPv6 over BLE stack is shown in Section 5.2. Lastly, Section 5.3 compares the IPv6 over BLE communication stack to the existing IPv6 over IEEE 802.15.4 implementation of Contiki regarding memory consumption (Section 5.3.1), communication overhead (Section 5.3.2), energy consumption (Section 5.3.3) and interference susceptibility (Section 5.3.4).

5.1 Interoperability

This section shows the interoperability of my communication stack with other RFC 7668 conform IPv6 over BLE stack implementations. In particular, I check the interoperability with the Nordic Semiconductor nRF52 communication stack (described in Section 3.1.1).

Setup

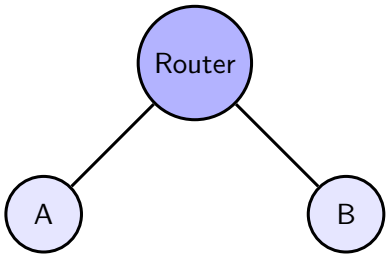

Figure 5.1: Network topology for evaluating the interoperability of the IPv6 over BLE implementation on the TI Sensortag (A: TI SensorTag, B: Nordic Semiconductor nRF52).

Figure 5.1 shows the network topology used to evaluate the stack's interoperability. A Texas Instruments SensorTag runs the IPv6 over BLE stack presented in this Thesis. A nRF52 DK from Nordic Semiconductor uses the IPv6 communication over a BLE link presented in Section 3.1.1. Both nodes are connected to the same BLE border router, a

Raspberry Pi 1 Model B with a LogiLink BT0015 BLE dongle (see Section 2.4.3) that is setup according to the installation guide in [38].

The nRF52 DK was programmed with a modified version of the existing Contiki `udp-server` implementation (`examples\ipv6\rpl-udp\udp-server.c`) that was adapted to support a maximum payload length of 1280 bytes and to use the User Datagram Protocol (UDP) port 61616. Hence, the nRF52 DK runs a UDP echo server that listens for UDP packets on UDP port 61616 and echos the received packets back to its sender. The SensorTag was programmed with a modified version of the `udp-client.c` application (`examples\ipv6\rpl-udp\udp-client.c`), a UDP client that periodically creates UDP packets of various size and tries to send these UDP packets to the UDP echo server. The client packets are sent every five seconds and use global IPv6 addresses as source and destination.

The request/response latency, the time between a UDP packet is sent and the corresponding UDP packet response was received, is measured at the UDP client using the `rtimer` of the Contiki OS.

Results

Figure 5.2 shows the average latency between the UDP packet sent by the client application and the corresponding UDP echo packet received by the client. The figure shows the average latency values over 20 consecutive measurements for every used packet length. The whole experiment was repeated three times.

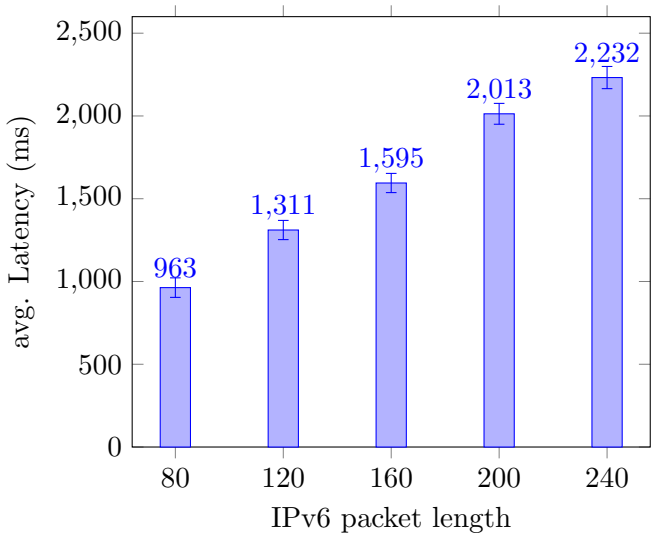

Figure 5.2: Diagram showing the average latency between UDP request and UDP response for different IPv6 packet lengths.

This experiment shows that the implemented IPv6 over BLE communication stack is interoperable with the Nordic Semiconductor nRF52 communication stack. Unfortunately, the communication between two BLE node devices has two major problems that cause interoperability issues. The first problem occurs when the SensorTag sends a UDP packet that is fragmented into several L2CAP fragments to the UDP echo server. When this

happens, both BLE connections of the border router(i.e., the connection to the SensorTag and to the nRF52 DK) terminate immediately without a proper BLE `disconnect` message. This behavior is dependent on the used L2CAP fragmentation size and only happens when the router needs to route packets from one node to another in the same IPv6 over BLE subnet. This problem is the main reason why the interoperability evaluation is only done with a maximum IPv6 packet length of 240 bytes, since the used L2CAP fragmentation size is currently 255 bytes.

Another problem is that the border router sends ICMPv6 redirect messages [42] whenever it needs to route IPv6 packets in the IPv6 over BLE subnet. The router falsely assumes that the best route between two IPv6 over BLE nodes in the same subnet is to directly talk to each other. Therefore the router sends ICMPv6 redirect messages to inform the node device about this direct route. Although this behavior is correct for IPv6 over IEEE 802.15.4 networks, it is not valid for IPv6 over BLE networks. As stated in Section 2.2.3, each BLE node is connected to the border router via an individual link. Any traffic between two nodes needs to be routed through the border router. Not only does the border router send redirect messages to node devices, it also has trouble to correctly route the packet to its destination, and each packet needs an additional hop to reach its correct destination node.

Figure 5.3 shows the ICMPv6 redirect messages and the incorrect routing behavior of the used border router. The network dump was created at the border router using `tcpdump`.

```
2001:db8::6ac9:bff:fe07:170d  ff02::1:ffda:7114                ICMPv6     96 Neighbor Solicitation for fe80::21a:7c
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6     96 Neighbor Advertisement fe80::21a:7dff:
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    ICMPv6     80 Echo (ping) request id=0xe000, seq=0,
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6    184 Redirect from 00:00:00:00:00:00:00:00
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    ICMPv6     80 Echo (ping) request id=0xe000, seq=0,
2001:db8::25a:11ff:fe62:4f61  ff02::1:ffda:7114                ICMPv6     96 Neighbor Solicitation for fe80::21a:7c
fe80::21a:7dff:feda:7114      2001:db8::25a:11ff:fe62:4f61    ICMPv6     96 Neighbor Advertisement fe80::21a:7dff:
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    ICMPv6     80 Echo (ping) request id=0x0000, seq=0,
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6    184 Redirect from 00:00:00:00:00:00:00:00
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    ICMPv6     80 Echo (ping) request id=0x0000, seq=0,
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    ICMPv6     80 Echo (ping) reply id=0x0000, seq=0, ho
fe80::21a:7dff:feda:7114      2001:db8::25a:11ff:fe62:4f61    ICMPv6    184 Redirect from 00:00:00:00:00:00:00:00
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    ICMPv6     80 Echo (ping) reply id=0x0000, seq=0, ho
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6    360 Redirect from 00:00:00:00:00:00:00:00
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    UDP       256 61616 → 61617  Len=192
fe80::21a:7dff:feda:7114      2001:db8::25a:11ff:fe62:4f61    ICMPv6    360 Redirect from 00:00:00:00:00:00:00:00
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    UDP       256 61616 → 61617  Len=192
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6    360 Redirect from 00:00:00:00:00:00:00:00
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    UDP       256 61616 → 61617  Len=192
fe80::21a:7dff:feda:7114      2001:db8::25a:11ff:fe62:4f61    ICMPv6    360 Redirect from 00:00:00:00:00:00:00:00
2001:db8::25a:11ff:fe62:4f61  2001:db8::6ac9:bff:fe07:170d    UDP       256 61616 → 61617  Len=192
fe80::21a:7dff:feda:7114      ff02::1                         ICMPv6    104 Router Advertisement
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
fe80::21a:7dff:feda:7114      2001:db8::6ac9:bff:fe07:170d    ICMPv6    360 Redirect from 00:00:00:00:00:00:00:00
2001:db8::6ac9:bff:fe07:170d  2001:db8::25a:11ff:fe62:4f61    UDP       256 61617 → 61616  Len=192
```

Figure 5.3: Network traffic captured on the border router that shows the falsely sent redirect messages and the routing problem of the used border router.

5.2 Network connectivity

This section shows the ability of the IPv6 over BLE communication stack to communicate with network devices outside the IPv6 over BLE subnet. To test network connectivity, a laptop outside of the subnet was used to ping the node device running the IPv6 over BLE stack.

Setup

The network topology used to test network connectivity is shown in Figure 5.4. A TI SensorTag runs the IPv6 over BLE connection stack presented in this Thesis. The SensorTag is connected to a Raspberry Pi with a BLE dongle (see Section 2.4.3) configured to be a border router [38]. A laptop is connected to the border router via Ethernet and is used to test the network connectivity of the SensorTag.

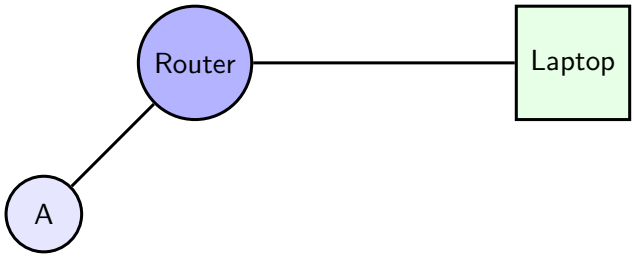

Figure 5.4: Network topology for evaluating the network connectivity of the IPv6 over BLE implementation (A: TI SensorTag)

To ensure that the SensorTag does not perform any interfering network activities, the SensorTag runs the `hello-world` example of Contiki. The laptop connected to the border router is used to send ICMPv6 echo requests to the SensorTag using the command line tool `ping`. The whole experiment was repeated three times.

Results

Figure 5.5 shows the latency of ICMPv6 echo request/response between the laptop executing the ping request and the IPv6 over BLE node for different IPv6 packet lengths. The latency values shown are the average values for 20 consecutive ICMPv6 echo request/response pairs performed for every packet length.

The measurements show that the latency between a request and its response is not linearly dependent from the used IPv6 packet length. Instead, it appears that the average latency depends on the number of L2CAP fragments used to transmit the IPv6 packet. As stated in Section 4.3.1, the current L2CAP fragment length is 255 bytes. Since IPv6 packets with a length of 80 bytes and 160 bytes fit into a single L2CAP fragment, the measured latency for both packet lengths is approximately the same.

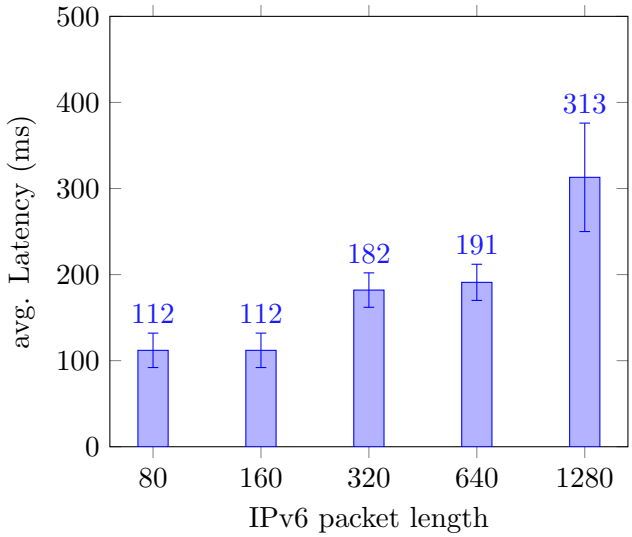

Figure 5.5: Diagram showing the average latency of an ICMPv6 echo request/response from the laptop to the SensorTag.

5.3 Comparing BLE to IEEE 802.15.4

This section compares the created IPv6 over BLE stack to the existing IPv6 over IEEE 802.15.4 stack of the Contiki OS. Section 5.3.1 compares the memory consumption of both communication stacks. Section 5.3.2 shows the communication overhead of IPv6 over BLE and IPv6 over IEEE 802.15.4 sending the same information. Section 5.3.3 compares the energy consumption of the standard IPv6 over IEEE 802.15.4 and the IPv6 over BLE communication stack executed on a TI SensorTag. Lastly, Section 5.3.4 compares the packet reception rate of both communication stacks under Wi-Fi interference.

Experimental setup

The IPv6 over BLE communication stack used in this evaluation section is in detail described in Section 4.3. The used BLE connection parameters: `connection interval`, set to 70 ms, and `slave latency`, set to 0, were defined by the border router at connection setup. The border router also configured the used BLE connection channels. The transmission power was set to `0 dBm`.

The IPv6 over IEEE 802.15.4 stack used for the following measurements is the existing implementation of the Contiki OS. This stack uses the IEEE 802.15.4 link layer implementation on channel 22 and transmits with a power of `0 dBm`. `ContikiMAC` with a `channel check rate` of 16 is used as the RDC layer implementation, `CSMA` as the MAC layer, and `sicslowpan` as the network layer. The only minor change to the IPv6 over IEEE 802.15.4 stack is the `QUEUEBUF_CONF_NUM` parameter was changed from 8 to 14. Without this change the maximum payload of 1280 bytes would not have been supported.

The Contiki application used is a modified version of the `udp-client` application (`examples\ipv6\rpl-udp\udp-client.c`) that was adapted to support a maximum payload of 1280 bytes and to use the UDP ports 61616 and 61617 as a server and client port,

respectively. The only difference between the applications, using either IEEE 802.15.4 or BLE as a link layer, is the configuration of the Contiki network stack in the configuration file `project-conf.h`.

5.3.1 Memory consumption

Figure 5.1 shows the memory consumption of the standard IPv6 over IEEE 802.15.4 communication stack and compares it to the memory consumption of the IPv6 over BLE communication stack that was created for this thesis. The only difference between the two applications is the used network stack configuration.

Table 5.1: Memory consumption of the IPv6 over IEEE 802.15.4 and the IPv6 over BLE communication stack of the Contiki OS. The memory consumption is divided into memory needed for code, constant data and global or static variables.

	IPv6 over IEEE 802.15.4	**IPv6 over BLE**
Code	58.52 kB	54.52 kB
Constant data	1.53 kB	1.5 kB
Global/static variables	13.76 kB	17.44 kB
Sum	**73.8 kB**	**73.2 kB**

The overall memory consumption of the IPv6 over BLE communication stack is lower than the memory needed for IPv6 over IEEE 802.15.4. Nevertheless, the IPv6 over BLE stack needs more global and static memory than the existing stack. Most of this static global and static memory is allocated by the L2CAP two fragmentation buffers (each having a length of 1280 bytes) and by the `rx_data_queue` (storing up to 1360 bytes) and `tx_data_queue` (storing up to 2160 bytes) used in the BLE-HAL implementation.

5.3.2 Communication overhead

Figure 5.6 compares the communication overhead of the IPv6 over IEEE 802.15.4 communication stack to the IPv6 over BLE implementation. For this test, the node sent UDP packets with different payload lengths to the border router and the actual number of transmitted byte was measured (e.g., to transmit an IPv6 packet with a length of 80 byte to the border router over the BLE link layer, only 44 byte of BLE data were sent). The exchanged packets were UDP packets using the link local IPv6 addresses as source and destination address and the UDP ports 61616 and 61617.

Figure 5.6 shows that the IPv6 over BLE stack needs fewer bytes to transmit the same IPv6 packet than the IPv6 over IEEE 802.15.4 implementation. The results indicate that BLE only needs 70% to 80% of the bytes that IEEE 802.15.4 needs to transmit the same information. The main reason for the smaller communication overhead is the smaller header size of BLE compared to IEEE 802.15.4. While IEEE 802.15.4 uses a header of 21 bytes to transmit a link layer payload of 80 bytes, BLE only needs 6 header bytes (3 link layer packets with a header of 2 bytes) to transmit the same link layer payload.

The number of bytes transmitted via BLE could even be more reduced, if the used L2CAP fragmentation size of the BLE-MAC layer would be increased.

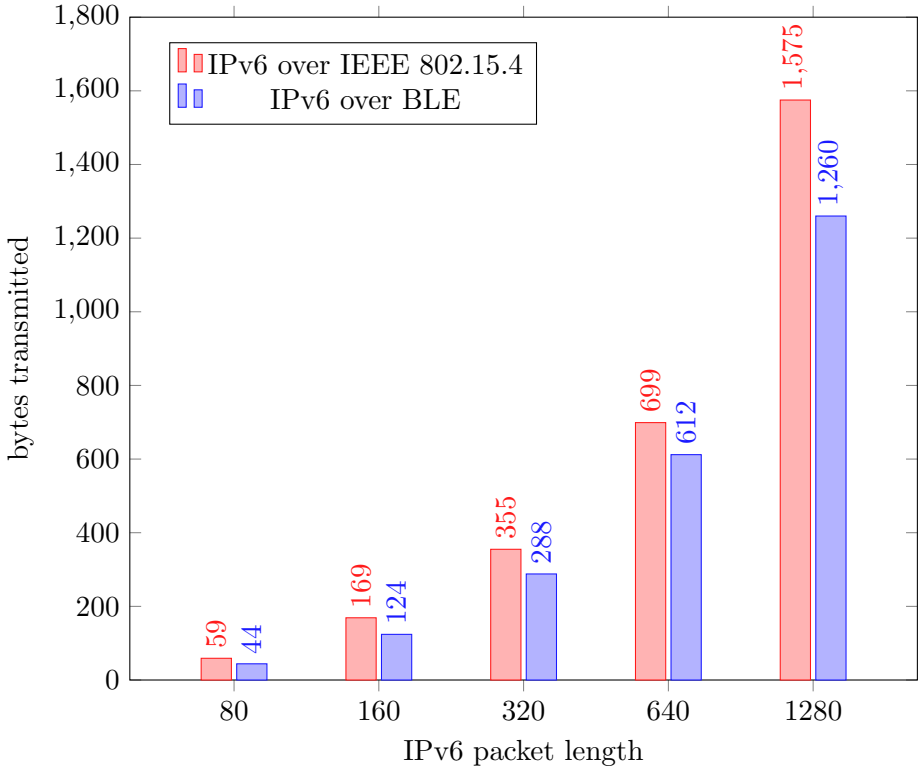

Figure 5.6: Comparison of the actual number of bytes transmitted by both communication stack implementations to send IPv6 packets with different lengths.

5.3.3 Energy consumption

This section gives a first estimate of the energy consumption of the IPv6 over BLE connection stack compared to the energy consumption of the existing IPv6 over IEEE 802.15.4 connection stack of the Contiki OS.

Detailed setup

Figure 5.7 shows the network topology used for measuring the energy consumption of both communication stack implementations.

In both cases (IPv6 over BLE and IPv6 over IEEE 802.15.4) the same SensorTag is used as a node device. The server used for BLE as a link layer is the Raspberry Pi border router described in Section 2.4.3. In order to measure the energy consumption of the IEEE 802.15.4 layer, another SensorTag was used as server, since the Raspberry Pi border router does not support `ContikiMAC`. During these experiments the node and the server had an approximate distance of 1 m.

Both server devices implement a UDP echo server that waits for UDP packets on port 61616. Every time a UDP packet is received, the server prints the UDP payload onto the console, and immediately returns a UDP packet containing the received UDP payload.

The node device runs a Contiki application that sends UDP packets to and receives packets from the server. This node is configured to use either the IPv6 over BLE or

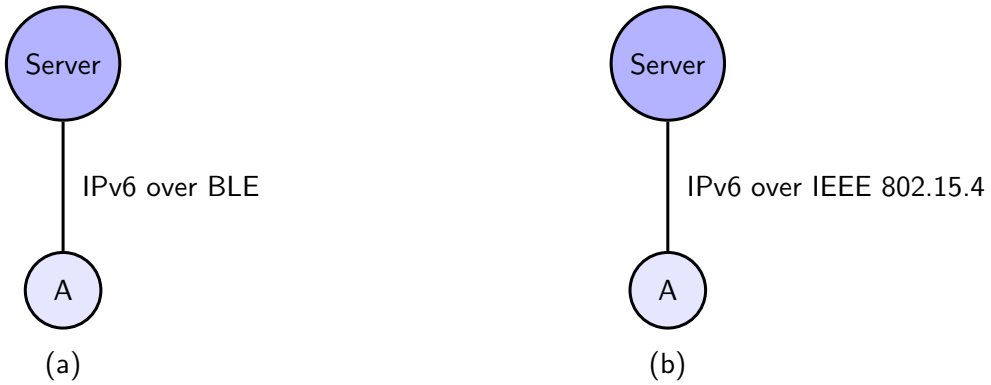

Figure 5.7: Network topology for evaluating the energy consumption of IPv6 communication over a BLE link (a) compared to a IEEE 802.15.4 link (b)

the IPv6 over IEEE 802.15.4 communication stack to communicate with one of the UDP servers. First, the application initializes the communication stack of the SensorTag and waits until the connection between node and server is established. This connection is checked by sending ICMPv6 echo requests to the server and waiting for the corresponding ICMPv6 echo response. After the connection is successfully established, the node waits for 5 seconds before sending the first UDP packet to the server. The UDP packets are sent every second to the server and use the link local addresses as source and destination.

Figure 5.8: Energy measurement setup consisting of a Raspberry Pi 2 model B connected to a TI LMP 92064.

The SensorTag is powered by an external power supply using its connectors for external battery packs. The energy consumption of the SensorTag is measured using a TI LMP92064 connected to a Raspberry Pi 2 model B (see Figure 5.8). This measurement

setup simultaneously samples the supply voltage and current with 12-bit resolution each at a sampling rate of 20kHz. The energy consumption of the SensorTag is calculated by integrating the measured voltage and current over time.

Result

Figure 5.9 shows the energy consumption of both communication stacks for different packet lengths. The energy measurement was started before the first UDP packet was sent and stopped after the 100th UDP packet was transmitted. Hence, Figure 5.9 shows the energy consumption of the TI SensorTag for exchanging 100 UDP request/response pairs over a time of approximately 100 seconds. Both evaluated link layers had a packet reception rate of 100% in this experiment.

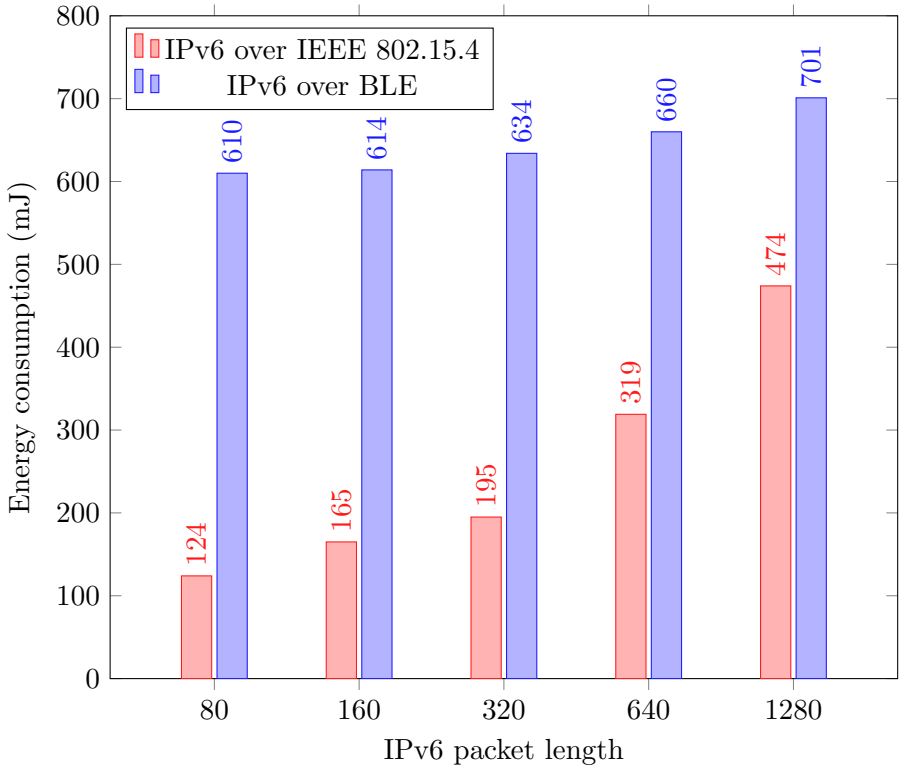

Figure 5.9: Comparison of the energy consumption of both communication stack implementations. The diagram shows the energy consumed by the TI SensorTag while sending 100 UDP request/response pairs over a time of 100 seconds.

Despite the expectation that the IPv6 over BLE communication stack should consume less energy than the existing IPv6 over IEEE 802.15.4 stack, Figure 5.9 shows that the energy consumption of the BLE link layer is higher for every evaluated IPv6 packet length. The measurements indicate that the energy consumption of IPv6 over BLE is less dependent on the amount of data sent than the energy consumption of IPv6 over IEEE 802.15.4.

One reason for the high energy consumption of the IPv6 over BLE communication

stack could be that the default BLE connection parameters (connection interval and slave latency) of the border router were used. Because node and router need to exchange BLE packets every connection event (every 70 ms for this experiment), both devices need to send and receive BLE connection packets, although no data needs to be exchanged for most of the time.

By increasing either the connection interval or the slave latency, the energy efficiency of the IPv6 over BLE communication stack could be optimized. Unfortunately, it was not possible to change the connection parameters on the Raspberry Pi border router used for this evaluation.

5.3.4 Interference susceptibility

This section compares the packet reception rate of the IPv6 over BLE communication stack to the existing IPv6 over IEEE 802.15.4 communication stack in the presence of Wi-Fi interference.

Setup

The setup of the interference susceptibility measurement is shown in Figure 5.1. Setup (a) shows the SensorTag (represented by A) connected via IPv6 over BLE to the Server. Setup (b) shows the same SensorTag connected to Server via IPv6 over IEEE 802.15.4, using IEEE 802.15.4 physical channel 22. This experiment uses the same border router/server setup as the energy measurement experiment in Section 5.3.3. The server and the client have an approximate distance of 1m.

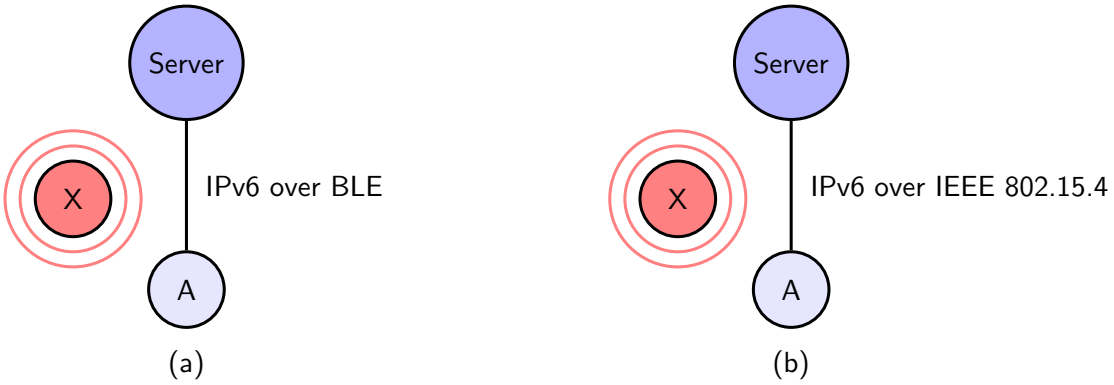

Figure 5.10: Network topology for evaluating interference susceptibility of IPv6 over a BLE link (a) compared to a IEEE 802.15.4 link (b)

A TmoteSky hardware platform running the JamLab [7] is used as the source of interference. JamLab is configured to emulate Wi-Fi traffic on IEEE 802.15.4 channel 22 with a transmission power of `0 dBm`. The jammer is either used in the mode `WiFi 1` (simulates light Wi-Fi traffic comparable to audio streaming) or in mode `WiFi 4` (simulates heavy Wi-Fi traffic comparable to a file download and simultaneous video streaming). The interference source is places in a distance of approximately 1m of the client and the server.

At first, the client sends ICMPv6 echo requests to the server to check if a connection was successfully established. After an ICMPv6 echo response is received, the client sends 100 UDP packets to the server, using a delay of 1 second between two consecutive packets.

Results

Figure 5.11 shows the packet reception rate of IPv6 over BLE and IPv6 over IEEE 802.15.4 under light Wi-Fi interference. According to the experiment, the IPv6 over BLE communication stack has a packet reception rate of 100% while lightly interfered. The IPv6 over IEEE 802.15.4 communication stack is able to transmit all packets with an IPv6 packet length of up to 320 bytes, although it does not perform channel hopping.

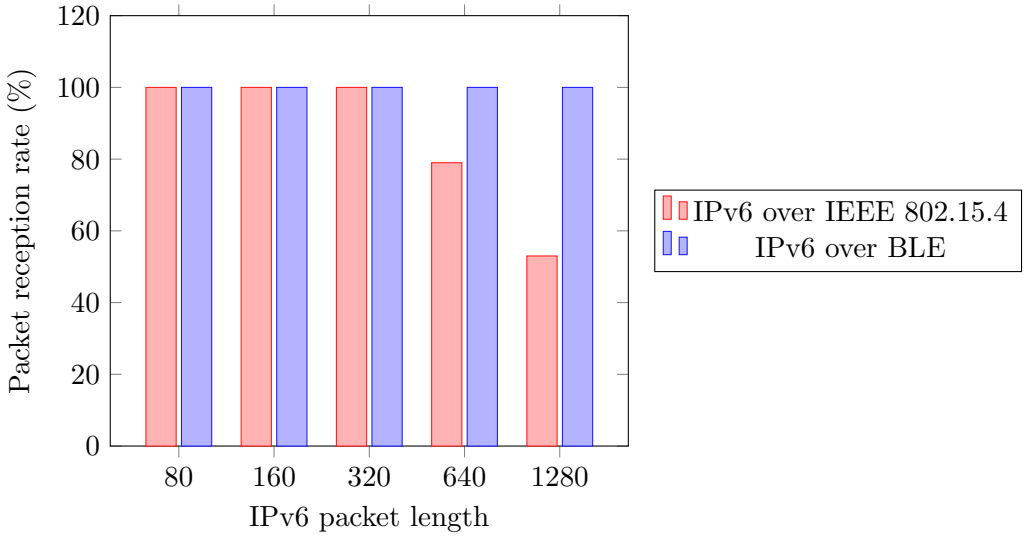

Figure 5.11: Packet reception rate of both communication stacks under light Wi-Fi interference (audio streaming).

Figure 5.12 shows the packet reception rate of both communication stacks under heavy Wi-Fi interference and a more significant difference in reliability. This experiment shows that, as expected, the IPv6 over BLE stack provides a better reliability than IEEE 802.15.4 even under heavy interference.

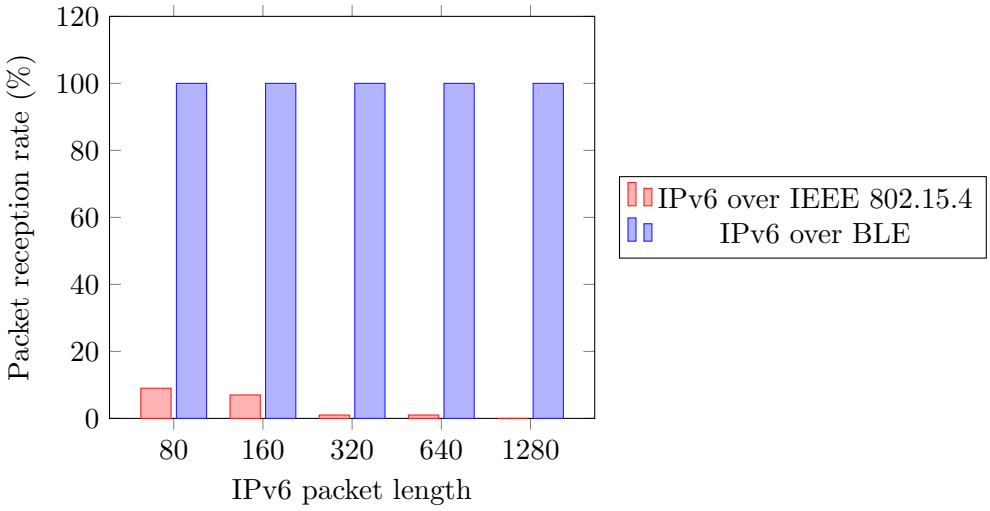

Figure 5.12: Packet reception rate of both communication stacks under heavy Wi-Fi interference (file download and simultaneous video streaming).

Chapter 6

Conclusions

Although BLE as a link layer for constrained wireless devices gets more attention in research and industry, no open source implementation of an IPv6 over BLE communication stack is currently available. Studies that evaluate BLE for wireless networks and the IoT usually compare the link layer capabilities of the different link layers available, but no study was published that compares IPv6 over BLE to IPv6 over IEEE 802.15.4 communication, even less on the same hardware platform.

This Master Thesis discusses the advantages of BLE as a link layer for constrained devices and summarizes the benefits of an open source implementation of such a communication stack and its impact on further research. The major contribution of this work is the design of an IPv6 over BLE communication stack that fits the architecture of the Contiki OS and its implementation for the TI CC2650 SensorTag hardware platform.

The experiments performed during this thesis provide a first comparison of the implemented IPv6 over BLE stack to the existing IPv6 over IEEE 802.15.4 communication stack of Contiki using ContikiMAC as its MAC layer implementation. This first evaluation shows that the implemented stack is interoperable with RFC 7668 [34] compliant border routers. The IPv6 over BLE stack needs less memory and has a smaller communication overhead than Contiki's stack using IEEE 802.15.4 and ContikiMAC. First measurements of the energy consumption show that the current BLE link layer implementation consumes more energy than the IEEE 802.15.4 implementation used by Contiki of the TI SensorTag. Nevertheless, the measurements indicate that the consumed energy of BLE is less dependent on the amount of data transmitted than the consumption of the existing stack. The evaluation of the interference susceptibility shows that the adaptive channel hopping performed by the BLE link layer results in a maximum packet reception rate of 100% even under heavy Wi-Fi interference and suggests a high reliability of the implemented communication stack.

Chapter 7

Future work

BLE stack. The current implementation of the IPv6 over BLE stack on the TI SensorTag may benefit from further improvements, especially in the BLE-HAL implementations. The scheduling of BLE connection events in the BLE-HAL could be optimized so that the application processor of the SensorTag would only wake up immediately before the next connection event needs to be scheduled. Hence, the application core would be more idle and the communication stack would be more energy efficient. It is also possible to extend the functionality of the already implemented BLE layer to fully support all BLE features specified in the Bluetooth Specification [3], such as GAP, GATT, ATT and SM.

Border router. The SensorTag could be connected to a Raspberry Pi via the slip radio interface and could be used as a border router device for IPv6 over BLE subnets. Such a BLE specific border router may resolve interoperability issues with the currently used Raspberry Pi border router.

Energy consumption. The energy efficiency and reliability of the IPv6 over BLE could be checked using different BLE connection parameters and interference scenarios. It would be interesting to compare the reliability of the IPv6 over BLE communication stack to the reliability of an IPv6 over IEEE 802.15.4 communication stack that uses MAC layer implementations performing frequency hopping (e.g., TSCH, MiCMAC).

Portability. Since the IPv6 over BLE communication stack was designed to be mostly hardware independent, the communication stack could easily be ported to other BLE hardware platforms with Contiki support. One possible target platform is the nRF52 from Nordic Semiconduction [36] that is already supported by the Contiki OS. This hardware platform provides a BLE radio that supports the HCI of BLE.

RDC functionality. As discussed during the presentation of the IPv6 over BLE communication stack, different implementations of stack layers may be used. The current stack implementation does not include any duty cycling functionality in the RDC layer. Additional RDC layer implementations could adaptively change the connection parameter of the underlying BLE data connection depending on the amount of data to be exchanged. During busy communication phases, the connection interval could be shortened to quickly

exchange the data. By increasing the connection interval and the slave latency of the BLE data connection, the RDC implementation would set the whole communication stack into a more energy efficient mode during communication phases with low or no data throughput. Another possible RDC layer implementation could disconnect the BLE connection in times where no data needs to be exchanged and only enable the BLE data connection for short amount of times to exchange IPv6 data before disconnecting again.

Advertising primitives. Lastly, the advertising privitives (advertising and scanning) of BLE could be used to exchange IPv6 packets between network devices. Currently these primitives are only used to setup the BLE data connection between node device and border router. Future work may create an RDC layer implementation that uses BLE advertising to directly send IPv6 packets to the border router and receives packets using BLE scanning. Such an IPv6 packet exchange on the advertising channels does not require any communication setup between node and border router or any periodic BLE packet exchange (i.e., BLE connection events). Therefore, it is very likely that a communication stack using advertising and scanning primitives to exchange IPv6 packets is even more energy efficient than existing stack implementations.

Bibliography

[1] Mats Andersson. Use case possibilities with Bluetooth Low Energy in IoT applications. *White Paper*, 2014.

[2] SIG Bluetooth. Bluetooth - Our History. `http://www.bluetooth.com/Pages/History-of-Bluetooth.aspx`. Accessed: 2015-09-18.

[3] SIG Bluetooth. Specification of the Bluetooth System - Covered Core Package version: 4.1. `https://www.bluetooth.org/en-us/specification/adopted-specifications`, December 2013.

[4] SIG Bluetooth. Internet Protocol Support Profile - Bluetooth Specification version: 1.0.0. `https://www.bluetooth.org/en-us/specification/adopted-specifications`, December 2014.

[5] SIG Bluetooth. Specification of the Bluetooth System - Covered Core Package version: 4.2. `https://www.bluetooth.org/en-us/specification/adopted-specifications`, December 2014.

[6] SIG Bluetooth. Supplement to the Bluetooth Core Specification - CSS version: 6. `https://www.bluetooth.org/en-us/specification/adopted-specifications`, July 2015.

[7] Carlo Alberto Boano, Thiemo Voigt, Claro Noda, Kay Römer, and Marco Antonio Zúñiga. JamLab: Augmenting Sensornet Testbeds with Realistic and Controlled Interference Generation. In *Proceedings of the* 10^{th} *IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 175–186. IEEE, April 2011.

[8] Wojciech Bober. Add support for nRF52 DK platform. `https://github.com/contiki-os/contiki/pull/1469`, 2016. Accessed: 2016-08-26.

[9] Philippe Bonnet, Allan Beaufour, Mads Bondo Dydensborg, and Martin Leopold. Bluetooth-based Sensor Networks. *SIGMOD Rec.*, 32(4):35–40, December 2003.

[10] Paulo Borges. Blessed. `https://github.com/pauloborges/blessed`, 2016. Accessed: 2016-08-25.

[11] B. Carpenter, Univ. of Auckland, S. Jiang, and Ltd Huawei Technologies Co. RFC 7136 - Significance of IPv6 Interface Identifiers. `https://tools.ietf.org/html/rfc7136`, February 2014.

[12] Varat Chawathaworncharoen, Vasaka Visoottiviseth, and Ryousei Takano. Feasibility Evaluation of 6LoWPAN over Bluetooth Low Energy. *CoRR*, abs/1509.06991, 2015.

[13] RS Components. Raspberry Pi - Model B. `http://docs-europe.electrocomponents.com/webdocs/127d/0900766b8127da4b.pdf`, 2016. Accessed: 2016-08-25.

[14] Cypress Semiconductor Corporation. Bluetooth Low Energy (BLE) Connectivity Solutions. `http://www.cypress.com/file/298446/download`, 2016. Accessed: 2016-08-25.

[15] Internet Architecture Board D. Thaler. RFC 4903 - Multi-Link Subnet Issues. `https://tools.ietf.org/html/rfc4903.html`, June 2007.

[16] A Dementyev, S Hodges, S Taylor, and J Smith. Power Consumption Analysis of Bluetooth Low Energy. *ZigBee and ANT Sensor Nodes in a Cyclic Sleep Scenario*, 2013.

[17] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. *SICS Technical Report*, 2011.

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

[19] The Apache Software Foundation. Mynewt. `http://mynewt.apache.org/`, 2015. Accessed: 2016-06-08.

[20] The Apache Software Foundation. NimBLE Introduction. `http://mynewt.apache.org/network/ble/ble_intro/`, 2015. Accessed: 2016-06-08.

[21] BlueKitchen GmbH. BlueKitchen - BTstack. `http://bluekitchen-gmbh.com/`, 2015. Accessed: 2016-06-08.

[22] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of Bluetooth Low Energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[23] Naresh Gupta. *Inside Bluetooth Low Energy*. Artech house, 2013.

[24] Robin Heydon. *Bluetooth Low Energy: The Developer's Handbook*. Prentice Hall, 2013.

[25] R. Hinden, Nokia, S. Deering, and Cisco Systems. RFC 4291 - IP Version 6 Addressing Architecture. `https://tools.ietf.org/html/rfc4291`, February 2006.

[26] Texas Instruments Incorporated. Texas Instruments - Bluetooth Low Energy Software Stack. `http://www.ti.com/tool/ble-stack`, 2016. Accessed: 2016-06-08.

[27] Texas Instruments. CC13xx, CC26xx SimpleLink Wireless MCU Technical Reference Manual. `http://www.ti.com/lit/ug/swcu117f/swcu117f.pdf`. Accessed: 2016-09-05.

[28] Texas Instruments. CC2650 SimpleLink Multistandard Wireless MCU. `http://www.ti.com/lit/ds/symlink/cc2650.pdf`. Accessed: 2016-02-05.

[29] Ed. J. Hui, Arch Rock Corporation, P. Thubert, and Cisco. RFC 6282 - Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. `https://tools.ietf.org/html/rfc6282`, September 2011.

[30] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pages 46–51. IEEE, 2007.

[31] Martin Leopold, Mads Bondo Dydensborg, and Philippe Bonnet. Bluetooth and Sensor Networks: A Reality Check. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 103–113, New York, NY, USA, 2003. ACM.

[32] G. Montenegro, Microsoft Corporation, N. Kushalnagar, Intel Corp, J. Hui, D. Culler, and Arch Rock Corp. RFC 4944 - Transmission of IPv6 Packets over IEEE 802.15.4 Networks. `https://tools.ietf.org/html/rfc4944`, September 2007.

[33] PrithviRaj Narendra, Simon Duquennoy, and Thiemo Voigt. BLE and IEEE 802.15.4 in the IoT: Evaluation and Interoperability Considerations. *Proceedings of the International Conference on Interoperability in IoT (EAI InterIoT 2015)*, October 2015.

[34] J. Nieminen, T. Savolainen, M. Isomaki, Nokia, B. Patil, AT&T, Z. Shelby, Arm, and C. Gomez. RFC 7668 - IPv6 over BLUETOOTH(R) Low Energy. `https://tools.ietf.org/html/rfc7668`, October 2015.

[35] The Institute of Electrical and Inc. Electronics Engineers. IEEE 802.15 WPAN Task Group 1 (TG1). `http://www.ieee802.org/15/pub/TG1.html`. Accessed: 2016-01-04.

[36] Nordic Semiconductor. Nordic Semiconductor Infocenter - nRF52 Series. `http://infocenter.nordicsemi.com/index.jsp`. Accessed: 2016-06-06.

[37] Nordic Semiconductor. Nordic Semiconductor IPv6 over Bluetooth Smart. `http://www.nordicsemi.com/eng/News/News-releases/Product-Related-News/Nordic-Semiconductor-IPv6-over-Bluetooth-Smart-protocol-stack-for-nRF51-Series-SoCs-enables-small-low-cost-ultra-low-power-Internet-of-Things-applications`. Accessed: 2016-02-01.

[38] Nordic Semiconductors. nRF5 IoT SDK - v0.9.0. `https://developer.nordicsemi.com/nRF5_IoT_SDK/doc/0.9.0/html/a00087.html`. Accessed: 2016-02-22.

[39] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011.

[40] Matti Siekkinen, Markus Hiienkari, Jukka K Nurminen, and Johanna Nieminen. How low energy is Bluetooth Low Energy? Comparative measurements with Zigbee/802.15. 4. In *Wireless Communications and Networking Conference Workshops (WCNCW), 2012 IEEE*, pages 232–237. IEEE, 2012.

[41] IEEE Computer Society. IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). `https://standards.ieee.org/about/get/802/802.15.html`. Accessed: 2016-02-09.

[42] IBM T. Narten and, E. Nordmark, Sun Microsystems, W. Simpson, Daydreamer, H. Soliman, and Elevate Technologies. RFC 4861 - Neighbor Discovery for IP version 6 (IPv6). `https://tools.ietf.org/html/rfc4861`, September 2007.

[43] Thingsquare. Contiki hardware. `http://www.contiki-os.org/hardware.html`. Accessed: 2016-06-06.

[44] Thingsquare. Contiki: The Open Source OS for the Internet of Things. `http://www.contiki-os.org`. Accessed: 2016-02-02.

[45] J.P. Vasseur and A. Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Elsevier Science, 2010.

[46] Ed. Z. Shelby, Sensinode, S. Chakrabarti, Ericsson, E. Nordmark, Cisco Systems, C. Bormann, and Universitaet Bremen TZI. RFC 6775 - Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). `https://tools.ietf.org/html/rfc6775`, November 2012.