Master Thesis at the Department of Mathematics and Computer Science of Freien Universität Berlin

Analysis and Comparison of Embedded Network Stacks

Design and Evaluation of the GNRC Network Stack

M. Lenders
Matrikelnummer: 4206090
mlenders@inf.fu-berlin.de

Advisor: Dr. Emmanuel Baccelli
Second Supervisor: Univ.-Prof. Dr. Jochen Schiller

Berlin, 19th of April 2016

Embedded network stacks are at the core of every software solution for the Internet of Things (IoT), since they provide access to the outside world. This thesis presents the proceedings of the design and implementation of the GNRC network stack. Furthermore, it compares this stack to other stacks with similar feature sets, namely lwIP [20] and emb6 [39]. I describe their functionality and architecture and provide an experimental quantitative evaluation based on the RIOT operating system [4, 3]. Since the term IoT is only defined very broadly, I also provide a definition for my view on the IoT and present the protocol suite used by both GNRC and the two reference stacks.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

19th of April 2016

_____

M. Lenders

Contents

List of abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
6Lo-ND	Neighbor Discovey Optimization for 6LoWPAN
6LBR	6LoWPAN Border Router
6LN	6LoWPAN Node
6LR	6LoWPAN Router
ACM	Association for Computing Machinery
API	Application Programmer Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CBOR	Concise Binary Object Representation
CoAP	Constrained Application Protocol
DAO	Destination Advertisement Object
DCCP	Datagram Congestion Control Protocol
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DNS	Domain Name System
DODAG	Destination-Oriented Directed Acyclic Graph
ETSI	European Telecommunications Standards Institute
FOSS	Free and Open Source Software
HAL	Hardware Abstraction Layer
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
ICMPv6	Internet Control Message Protocol, version 6

ICN	Information-centric networking
IPC	Inter-process Communication
IT	Information Technology
IP	Internet Protocol
IPv4	Internet Protocol, Version 4
IPv6	Internet Protocol, Version 6
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
L2	Link Layer
loc	lines of code
MAC	Medium Access Control
MIME	Multipurpose Internet Mail Extensions
MLD	Multicast Listener Discovery
MMU	Memory Management Unit
MTU	Maximum Transmission Unit
NDP	Neighbor Discovery Protocol
REST	Repesentational State Transfer
OSPF	Open Shortest Path First
PPP	Point-to-Point Protocol
RPL	Routing Protocol for Low-Power and Lossy Networks
SCTP	Stream Control Transmission Protocol
SICS	Swedish Institute of Computer Science
TCP	Transmission Control Protocol
TDMA	Time division multiple access
TFTP	Trivial File Transfer Protocol
TSCH	Time-Slotted Channel Hopping
UDP	User Datagram Protocol
WWW	World Wide Web
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Networks
XML	Extensible Markup Language

Acknowledgment

First and foremost, I would like to thank my advisor Dr. Emmanuel Baccelli. He always had an open ear to my questions, provided me with fair criticism on the ongoing status of my thesis, and always encouraged me to let my contributions show more prominently.

I would also like to thank the RIOT community and my colleagues for valuable input on the topic, my thesis, and their acceptance of my personality. I would like to thank Jakob Pfender especially for proof-reading my thesis and his patience with my horrible English grammar. Furthermore, I am grateful for Hauke Petersen's contributions to GNRC's design and constant advice and criticism regarding my work. On the same note I would like to thank Oliver Hahm and Kaspar Schleiser. Without their input the API of GNRC probably wouldn't be as accessible as it is today. I would also like to thank Jose Alamos, Michael Anderson, Ken Bannister, Simon Brummer, Thomas Eichinger, Johann Fischer, Cenk Gündoğan, Nick van IJzendoorn, René Kijewski, Peter Kietzmann, Ludwig Knüpfer, Daniel Krebs, Martin Landsmann, Joakim Nohlgård, Francisco Javier Acosta Padilla, Andreas Pauli, Kévin Roussel, Aaron Sowry, Lotte Steenbrink, and Takuo Yonezawa for their contributions and advice to GNRC and RIOT's networking infrastructure in general.

Further acknowledgment has to go to Univ.-Prof. Dr. Jochen Schiller as the second supervisor of this thesis and the rest of the exam colloquium. I am grateful for their valuable comments on this thesis.

Finally, my gratitude goes to my family and friends for their unending support and encouragement during my years of study and especially throughout the time of research for and writing of this thesis. Special thanks must go to my parents and my best friend Carla, who were at my side in emotional heavy periods during these months. Without their guidance this accomplishment would not have been possible.

Thank you all!

Martine Lenders

This page was intentionally left blank

1 Introduction

1.1 Motivation

The *Internet of Things (IoT)* has a steadily growing industry [27, 62] and there are many visions for it [74]. Currently, most manufacturers use proprietary technologies such as ZigBee [95] and Z-Wave [92] – especially on the *smart home* market. But ever maturing standardization of open IP-based technologies [45] has also sparked new interest in open standards, not only in Free and Open Source Software (FOSS) communities, but also by bigger industry representatives, as the participation in the standardization process of recognizable names such as ARM, AT&T, Cisco, Ericsson, Intel, Microsoft, Nokia, and many more shows [45, 48, 64, 66, 76].

The IoT has the potential to change our lives and how we see and use the Internet and future innovations, much like the introductions of the World Wide Web, smartphones or other technologies have had. As such, I am of the opinion that for a sustainable future of the IoT, a further opening of the specifications and implementations is vital. Not only are closed solutions raising concerns regarding privacy and security [87], but an open standardization will also allow for participation from smaller businesses, private persons or communities interested in technology such as e.g. several FOSS communities, or the maker/hacker community. The presence of these – in economic terms – grassroots movements in the process will help to diversify and ultimately strengthen the field. This has already been seen with other technologies such as the traditional Internet or the changes the rapid prototyping industry saw with the advent of cheap and accessible 3D printers [81].

Several platforms that follow this Open Source spirit have emerged in the last years. Examples for software platforms are the operating systems Contiki [23], FreeRTOS [5], RIOT [4, 3] and TinyOS [57].

Since the idea of Open Source hardware is still very young [18], hardware platforms of this kind for the IoT are still rather rare, with the Arduino micro-controller [1] and the single-board computer series Raspberry Pi [70] currently being the most popular on the different ends of the hardware spectrum of embedded systems.

During the integration of components of 6LoWPAN – the Internet Engineering Task Force (IETF)'s adaptation protocol for IPv6 over IEEE 802.15.4 (see subsection 1.4.3) – into the previous network stack by S. Zeisberg [93] of the operating system RIOT (then still called μkleos) for my Bachelor Thesis [51], I quickly realized that its design did not really allow for new components to be easily integrated. This

trend would later also show for other students working on the integration of other components like O. Gesch's transport layer module „DEStiny“ [35] and E. Engel's RPL implementation [26]. When I finally made the decision to port the network stack originally developed on the MSB A2 platform [34] by Freie Universität Berlin (which has a sub-GHz CC11xx radio using a proprietary PHY protocol [82]) to the IEEE 802.15.4 based IoT LAB M3 Open Node [29] platform for the 1st ETSI 6LoWPAN plugtest [28], we realized that an overhaul of our networking structure was needed. Likewise, none of the existing solutions seemed to fit our needs (see section 1.5 for further details). This thesis documents both processes and results of the effort of creating this new networking architecture for the RIOT operating system.

1.2 Structure of This Thesis

The remainder of this introduction will outline the contributions of this thesis (section 1.3), attempt to isolate my understanding of the IoT and describe the suite of network protocols used in this thesis to implement IoT capabilities (section 1.4). By adding this to the introduction I am set to list other works related to this thesis in that context (see section 1.5).

In chapter 2 I describe the functionalities of the operating system I was working with for this thesis: RIOT. Chapter 3 then provides an in-depth look into the inner workings of the network stack GNRC, the main contribution of this thesis. Afterwards, I provide a brief overview of two alternative stacks – emb6 and lwIP – in chapter 4 and provide an evaluation of GNRC, both individually and in comparison to emb6 and lwIP, in chapter 5. Finally, I conclude my thesis in chapter 6 and provide an outlook regarding future works in this topic.

1.3 Contributions of This Thesis

In preparation and during the my work on this thesis I made several contributions not only to my working group at Freie Universität Berlin, but also to the community of the RIOT operating system which I would like to present in the following.

- Co-design of the GNRC network stack, described in chapter 3.
- Co-design of the network device driver API `netdev`, described in section 2.3.
- Design of the minimalist transport layer connectivity API `conn`, described in section 2.4.
- Design of the common integration API to `netdev` for IEEE-802.15.4-based devices.
- Documentation work on various components of RIOT, including
  - top-level overview documentation,

  - core functionalities,
  - third-party package management, and
  - networking capabilities.

- Preliminary steps towards a unified testing environment for RIOT applications (utilizing the unit-test library `embunit` and the python library `pexpect`).
- Preliminary port of RIOT's TAP-based network device on the `native` platform to the `netdev` API.
- Implementation of GNRC's IPv6 module, including
  - next-header demultiplexing,
  - preliminary source address determination,
  - next hop determination,
  - NDP address resolution, and
  - NDP router and prefix discovery.
- Implementation of GNRC's 6LoWPAN module, including
  - uncompressed packet handling,
  - fragmentation and reassembly of IPv6 datagrams,
  - compression of IPv6 headers, and
  - optimization of NDP for 6LoWPANs (6Lo-ND).
- Porting of the lwIP network stack (see section 4.2) to RIOT
- Porting of the emb6 network stack (see section 4.3) to RIOT
- Co-authorship in the publication "Old Wine in New Skins?: Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices" [69] to the 2015 Workshop on IoT challenges in Mobile and Industrial Systems (IoTSys'15, companion event of MobiSys'15).
- Presentation of said publication at IoTSys'15 in Florence.
- Submission of co-authored paper "RIOT: Reconsidering Operating Systems for Low-End IoT Devices" [36] to the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).
- Support, minor fixes and maintenance for implementations of other components of GNRC's and devices or modules implementing `netdev` or `conn`, including
  - GNRC's UDP module,
  - GNRC's TCP module,
  - GNRC's RPL module,
  - GNRC's capability to process IPv6 extension headers,
  - the `xbee` shield IEEE 802.15.4 driver,
  - the `at86rf2xx` IEEE 802.15.4 driver,
  - a DNS implementation based on `conn`, and
  - a TFTP implementation based on `conn`.

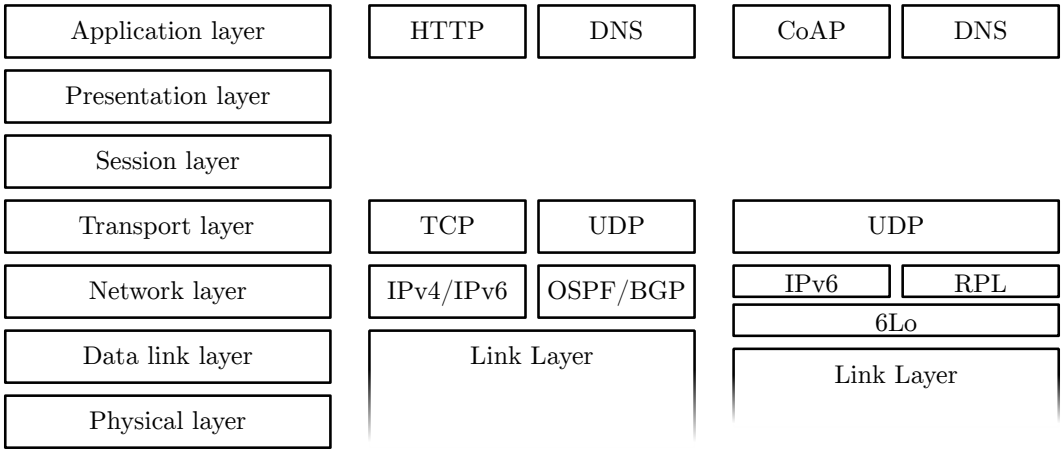


Figure 1.1: Comparison of network stack architectures

- Major contributions to the RIOT community in documentation, networking, system libraries and third-party package support, too numerous to name individually.

1.4 The Internet of Things

1.4.1 Definition

Used primarily as a non-descriptive buzzword, the term "Internet of Things (IoT)" has been circulating for a while now, and depending on who you ask, there will be a different understanding of what it means. As such, there is still no unifying definition, depending on what the people and organizations working with it are focusing on. In general, there are three perspectives [74] that either focus

- on *the Internet*,
- on *things*, or
- on *data*

The most unifying definition of the IoT I found is:

> An open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources, reacting and acting in face of situations and changes in the environment [59].

Resulting from this there exist multiple silo solutions today [16] that are connected via a cloud solution in the best case or not connected at all.

For the purpose of this thesis I will focus on a definition based on the Internet, specifically the protocol suite specified by the IETF. For the sake of simplicity, I

will refer to this suite as the *IoT stack* (see Figure 1.1c). Since it builds upon open and established Internet standards, I regard it as the most accessible solution.

The *IoT stack* tries to be as compatible as possible to the traditional TCP/IP protocol suite used in the conventional Internet, while honoring the constraints given for the IoT devices they are targeted at:

- **Large address space**: Many devices will join the IoT. For every human being on this planet the number of devices will probably be in the double-digit range, easily putting the number in the billions [27].
- **Energy requirements**: Nodes intended for the IoT are often required to run on battery and ideally for a long time – i.e. many years – since they might be in hard to reach places. As a result, most devices commonly also have the following constraints:
  - **Low processing power**: Most devices geared towards the IoT stack are microcontrollers and usually only have clock rates of a few MHz.
  - **Memory**: Likewise, the memory is only available in terms of KiB, or MiB at best.
  - **Lossy medium**: Most IoT technology is based around wireless communication, which is far less reliable in packet delivery rate than wired communication. To fulfill the previously mentioned energy requirements, many Link Layer (L2) technologies also only have very sporadic Medium Access Control (MAC) protocols with long node sleeping cycles.

The IETF classifies the kind of devices we are talking about as class 0 and class 1 devices [8].

Based on these constraints, the IETF basically worked their way up the traditional TCP/IP stack. The resulting IoT protocol suite can be seen in Figure 1.1 and is generally understood to be comprised of the 6LoWPAN adaptation layer (at least for IEEE 802.15.4), IPv6, UDP, and CoAP, which I will briefly summarize in the following sections.

1.4.2 IPv6

There are a number of reasons why the Internet Protocol, Version 6 (IPv6) is best suited for the IoT. First of all, we need to support an Internet Protocol (IP) to be able to speak with other TCP/IP stack implementations i.e. the traditional Internet. Furthermore, IPv6 provides the IoT with another advantage that the alternative – the Internet Protocol, Version 4 (IPv4) – can't provide: Its large 128-bit address space. As stated in subsection 1.4.1, the IoT will consist of billions of nodes and the 32-bit address space of IPv4 is already depleted [43]. Another advantage is IPv6's focus on multicast rather than broadcast which could be used to save energy[1].

[1]Not for IEEE 802.15.4 though, since it only supports broadcast.

Though IPv6 itself isn't any different from the traditional TCP/IP stack, the Neighbor Discovery Protocol (NDP) – which utilizes the Internet Control Message Protocol, version 6 (ICMPv6) – needed some adjustments (referred to as 6Lo-ND in the remainder of this thesis), largely because of the architecture of IEEE 802.15.4 Wireless Personal Area Networks (WPANs)s and the different address formats of IEEE 802.15.4, but also to disseminate compression contexts of 6LoWPAN throughout a WPAN.

1.4.3 6LoWPAN

IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) is an adaptation layer protocol which allows for IPv6 communication over IEEE 802.15.4 WPANs (the link layer of the ZigBee stack) [64]. Later adaptation layers based on 6LoWPAN for other L2 as ITU G.9959 [10] and Bluetooth low energy [66] were introduced and further adaptation layers are in the making (DECT ULE [60], MS/TP [58], and NFC [44] to name a few). These other adaptation layers often share the 6Lo prefix of 6LoWPAN (as designated in Figure 1.1) in their name. When talking about 6LoWPAN however, one usually refers to communication over IEEE 802.15.4. As this is the modus operandi for the most part during my implementation work, I will use 6LoWPAN as an example for all 6Lo protocols in the remainder of this thesis.

Besides allowing for the formation of mesh networks, IEEE 802.15.4, at least up until the 2011 version of the standard, is also only able to send frames of at most 127 bytes [49]. This is in direct contradiction to IPv6's required *minimum* Maximum Transmission Unit (MTU) of 1280 bytes [64].

To achieve transmission of IPv6 packets regardless of the size constraints of IEEE 802.15.4, 6LoWPAN provides a byte-optimized packet fragmentation (beyond the fragmentation provided by IPv6 itself) and both stateless and stateful header compression. For stateful compression, context information is disseminated throughout the network by NDP.

To utilize the mesh functionality of IEEE 802.15.4, 6LoWPAN provides a mesh addressing header and an associated broadcast header that can be appended to the frame to use the so called "mesh-under" communication. For "route-over" communication (i.e. to leave routing to a routing protocol, see subsection 1.4.4), this header can be skipped.

It achieves these different functionalities by using the first byte of the IEEE 802.15.4 frame payload as a dispatch to identify the type of header. Multiple header types can be strung after one another, but the order is defined:

1. Mesh addressing header
2. Broadcast header
3. Fragmentation header
4. Compression header

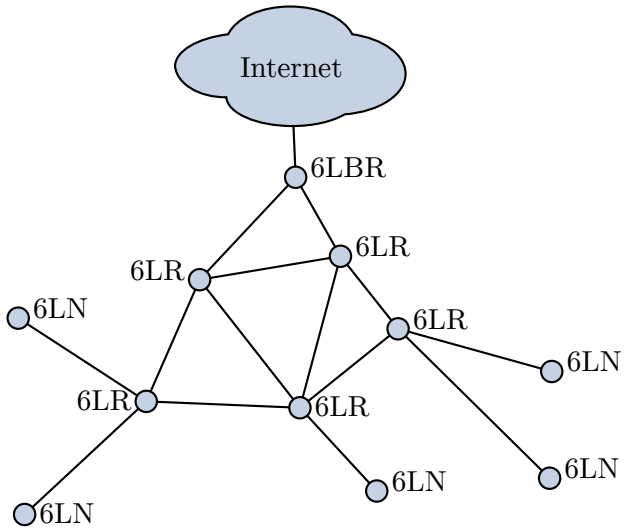


Figure 1.2: A 6LoWPAN route-over topology

As for the architecture of a 6LoWPAN network, in route-over mode (which I will use for my experiments) there are three node types (see Figure 1.2) [76]:

- **6LoWPAN Node (6LN):** a host or router in a WPAN.
- **6LoWPAN Router (6LR):** a router inside a WPAN that is able to forward IPv6 packets.
- **6LoWPAN Border Router (6LBR):** a router at the intersection between a WPAN and another network.

6LRs are not present in a mesh-under topology, but the other nodes are.

For a more in-depth description of the protocol, it is highly recommended to read the 6LoWPAN specifications [64, 41, 76], but for the purpose of this thesis, this summary will suffice.

1.4.4 RPL

The Routing Protocol for Low-Power and Lossy Networks (RPL) [89] can be used for multi-hop routing in IoT-subnets. Since the IoT as it is described here is closely related to Wireless Sensor Networks (WSN), it assumes the network to be target-oriented: There are multiple nodes in the subnet that primarily communicate using one higher-ranking router – the *"target"* (and usually the 6LBR; see subsection 1.4.3) – with the rest of the Internet. As such, the network is structured as a Destination-Oriented Directed Acyclic Graph (DODAG), a graph consisting of directed edges, without cycles, and with all paths leading to a single destination – the DODAG root (see Figure 1.3).

All communication for RPL is over ICMPv6, meaning RPL works over IPv6, but not IPv4. The message types send behave as follows:

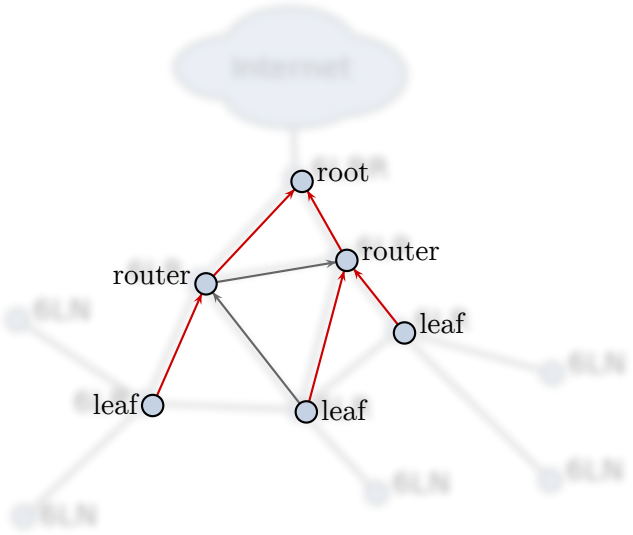


Figure 1.3: A RPL DODAG in the LoWPAN of Figure 1.2 with the red arrows representing preferred routes and gray arrows representing secondary routes (faded lines represent a working connection)

- All nodes in the network send **DODAG Information Objects (DIOs)** periodically to inform other nodes of their presence in a DODAG. Based on this information, nodes that join a network can choose the parent with the best route to the DODAG root. It is also possible for a node to request a DIO with a **DODAG Information Solicitation (DIS)**.
- To be able to take routes downstream (from the DODAG root to another node in the network) all nodes except the DODAG root send **Destination Advertisement Objects (DAOs)** to inform about the nodes they are able to reach or none if they are one of the most downstream nodes – a leaf.

Note that RPL leaves are not the same nodes as 6LN in a 6LoWPAN (see subsection 1.4.3), as 6LN don't have any forwarding capabilities, while leaves are forwarding routers that just don't have any children *in the RPL DODAG*. Non-forwarding 6LN hosts, while not being part of the DODAG itself, can connect to a leaf (or any other router in the DODAG for that matter) by just setting them as their default router.

A DODAG is usually identified by one of the RPL root's global unicast addresses (the DODAG ID).

1.4.5 UDP

As already stated in subsection 1.4.1 and subsection 1.4.3, most IoT L2 come with *large delays*, *high packet loss*, and *small payload lengths*. All of these make the Transmission Control Protocol (TCP) with its reliable and ordered data transmission and congestion and flow control mechanisms unsuitable for the IoT. Especially the

latter is a critical disadvantage compared to lossy networks: Congestion control *reduces* the retransmission rate, rather than increasing it since it assumes that packet loss is a congestion problem rather than an inherent property of the medium [63]. For these reasons, the far simpler *User Datagram Protocol (UDP)* is the preferred transport layer protocol for the IoT (though TCP is of course also possible, albeit discouraged).

Alternative transport layer protocols such as for example Datagram Congestion Control Protocol (DCCP) [46] or Stream Control Transmission Protocol (SCTP) [78] are also possible[2], but are currently not considered due to their low popularity.

Like IPv6, UDP has no modifications for the IoT (though 6LoWPAN provides header compression for it).

1.4.6 CoAP

The easiest way to describe the Constrained Application Protocol (CoAP) [77] is to view it as a binary version of the Hypertext Transfer Protocol (HTTP) over UDP (or other alternative transports, like DCCP or SCTP [77, section 3]). It is designed to be easily translatable to HTTP by providing the framework to implement a Repesentational State Transfer (REST) API.

Since UDP does not provide any reliability, CoAP provides this optionally via its messaging layer. Above the messaging layer the requests/response layer is used to implement the RESTful behavior.

As with HTTP, the way to represent the delivered data can be chosen freely for the most part and can e.g. be JavaScript Object Notation (JSON) [11], Extensible Markup Language (XML) [12], Concise Binary Object Representation (CBOR) [9], or even Hypertext Markup Language (HTML). But rather than using the string representation of the Multipurpose Internet Mail Extensions (MIME) type, CoAP uses a standardized sub-registry of numeric identifiers to identify the data representation format to reduce overhead in the CoAP headers [77, section 12.3].

For the scope of this thesis, I will mainly focus on the layers below and including the transport layer, so CoAP will not be mentioned as much in the remainder of this work. This section is only included to complete the full IETF IoT stack.

1.5 Related Work

The idea of a modular network architecture to both provide a clear interface and reduce code duplication (and in turn code size) certainly isn't a new idea at all. The UNIX STREAMS framework [71] already provided a very loosely coupled approach to modularity at the very beginning of the Internet, with data-handling modules

[2] They can even be used with CoAP (see subsection 1.4.6)

(including network protocol implementations) only communicating via input and output queues.

Later research into embedded networking such as for example with the *x*-Kernel [42] also picked up the idea of modularity to provide a micro-kernel specifically for development of network protocols on embedded systems.

In the modern IoT that I drafted out in section 1.4 we are not only faced with the constraints given by the hardware, but also with the fact that the range of hardware platforms is vastly heterogeneous. As such, most operating systems either decided to support only one platform – as for example mbed OS [2] (ARM) or LiteOS [14] (AVR) – or went for a *"one-fits-most"* approach as e.g. the previously mentioned Contiki [23], FreeRTOS [5], TinyOS [57], and of course RIOT [4, 3], which is the focus of this thesis.

Both TinyOS and Contiki come with their own network stack implementation: BLIP [67] and uIP [21] respectively. Both are implementations of the IoT stack presented in section 1.4. Furthermore, Contiki also comes with the more modular RIME [22] stack.

Outside of the operating system realm there are also some stand-alone network stacks. In this thesis I examined lwIP [20] (see section 4.2), a modular full-featured network stack, designed for low memory consumption, and emb6 [39] (see section 4.3), a fork of Contiki's uIP stack, that allows it to run without Contiki's proto-threads [24].

Due to their independence from operating systems, I even ported these two stacks to RIOT during my thesis to have a reference for comparison with GNRC. However, we decided against making either of them the default stack of RIOT for a number of reasons. lwIP e.g. did not have any 6LoWPAN support at the time when we started and still lacks RPL support (see section 4.2) and emb6 was too close to the paradigms of our old stack with all of the bad experiences we had had with it.

Other stacks we looked into were the NanoStack [50], which in later versions became proprietary, and the CiAO/IP stack [7], which focuses on aspect oriented-programming and is implemented in the AspectC++ dialect for this reason. Other stacks proved to be very use-case oriented [61, 72, 91] or very restricted and not extensible by design [17] and thus did not fit into our requirements (see subsection 3.2.1) at all.

1.6 Summary

This chapter explained the motivation for the design and development of an open source network stack for the IoT in that only free and open source software can provide the needs for privacy and security needed by the IoT and that only openness may lead to a well established technology. To set-up a frame of reference for the

related work I then settled on a definition of the IoT and described the constraints and requirements of this definition:

- A large address space and
- energy requirements that result in low processing power, low memory capabilities, and lossy transmission mediums.

I also specified that in the context of this thesis, the connectivity provided by protocol suite outlined by the IETF is what I mean by IoT. The protocols that make up this suite are the link-layer adaptation protocol 6LoWPAN to allow IPv6 traffic over IEEE 802.15.4, some optimizations to the NDP of IPv6, the routing protocol RPL, and the application layer protocol CoAP, which allows HTTP-like REST-communication over lossy transports like UDP. This allows for easy integration into the traditional TCP/IP Internet and the World Wide Web.

Finally, I put this thesis into broader context by providing an overview of some related work, both in the world of modular network stacks and network stacks implementing the IoT stack protocol suite. The idea of highly modularized networking architectures certainly isn't new, as solutions for this – even in the embedded world – go back to the 1980s. Established operating systems come with their own embedded solutions and platform independent solutions exist as well.

2 The RIOT Operating System

2.1 Introduction

The network stack designed in the context of this thesis was originally developed for the RIOT operating system. As such it is important to understand at least the basic operation and network related Application Programmer Interfaces (APIs) of this system.

RIOT is a real-time operating system for the IoT and was developed originally by Freie Universität Berlin, HAW Hamburg and INRIA. It has its roots in software previously provided by all three institutions. Today it is developed by a worldwide community under the LGPLv2.1 license.

The following chapter gives a short overview over RIOT's kernel and its three main components: the scheduler, the IPC, and its mutexes (section 2.2). Likewise, we look into other components specifically intended for networking access: the common network device API `netdev` (section 2.3) and the transport layer connectivity API `conn` (section 2.4). This will allow us to put the GNRC network stack described in chapter 3 into the perspective of the operating system it was developed for.

Lastly, I take a short look into the third-party package system of RIOT (section 2.5), which allowed me to integrate the other two stacks described in chapter 4 for comparison.

2.2 The Kernel

2.2.1 Overview

Traditionally, operating system kernels fall into two categories, although there are hybrid solutions today: *monolithic kernels* and *micro-kernels*. Monolithic kernels, such as for example the Linux kernel, include all of the functionality and drivers the operating system needs to run and manage the hardware it is running on. Micro-kernels, on the other hand, only include the most basic functionalities such as memory management, process and/or thread scheduling and basic Inter-process Communication (IPC). Everything else is provided by external modules that run in the same space as user software [80].

The RIOT kernel is a real-time capable micro-kernel [3]. Real-time capability here refers to *soft real-time* capabilities [37], as the RIOT scheduler guarantees a thread to deterministically keep a deadline, but there also will be no harmful

behavior if that deadline is not met [36]. As most platforms supported by RIOT don't offer a Memory Management Unit (MMU), the kernel only extends its memory management to assigning a memory stack to a thread on initialization, but to date offers no protection for that memory region. Additionally, the RIOT kernel also provides capabilities to synchronize memory access between threads via mutexes.

2.2.2 Scheduling

The RIOT scheduler uses a very simple tickless scheduling policy of complexity $O(1)$: The thread with the highest priority not blocked by given circumstances (e.g. waiting for IPC message) runs until interrupted by the Interrupt Service Routine (ISR). If all threads are blocked, a special *idle* thread is run, which allows the device to go into a low-power mode [3]. This way, the soft real-time capabilities are provided by using ISR triggers for event management. It also allows the operating system to hold the energy consumption requirements set by the IoT application realm (see subsection 1.4.1), as no periodic wake-ups are required as they would be needed for time-based scheduling.

2.2.3 Inter-process Communication

The IPC allows not only for blocking and non-blocking, but also for synchronous and asynchronous communication [52].

Messages consist of both a content field and a type field identifying the kind of content. In addition to this, they contain a PID field that identifies the sender.

Synchronous mode is the default mode of communication. To activate asynchronous communication, a thread needs to initialize its message queue; however, if this message queue is full, the kernel will fall back to synchronous communication.

In case of blocking communication, the thread will be blocked and the scheduler will run the next waiting thread in the thread queue. The functions for blocking communication are `msg_send()` and `msg_receive()`. Additionally, there is `msg_send_receive()` which allows the receiving thread to reply contextually with `msg_reply()`.

For non-blocking communication the message needs to be sent in asynchronous mode, otherwise the message will be dropped. To use non-blocking communication, a user must call the functions `msg_try_send()` and `msg_try_receive()`.

2.2.4 Synchronization Via Mutexes

To synchronize memory access external to a thread's memory stack between different threads, the RIOT kernel offers a simple mutex (mutual exclusion) scheme.

Mutexes can be locked and unlocked with `mutex_lock()` and `mutex_unlock()` respectively. `mutex_lock()` is blocking, while `mutex_unlock()` is not. As an alternative,

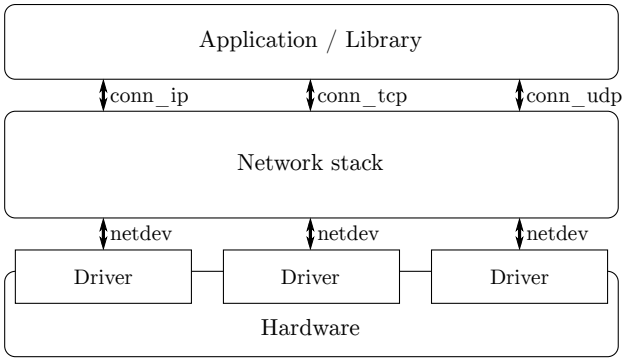


Figure 2.1: RIOT's networking architecture

the API also provides `mutex_unlock_and_sleep()`, which puts a thread to sleep after the mutex was unlocked, and the non-blocking locking function `mutex_trylock()`.

2.3 The Network Device API – netdev

netdev is the network device driver API used by RIOT to abstract a network device and allows access to network devices independent from the device itself. Thanks to this API, porting a network stack is rather simple, since its hardware abstraction only needs to be wrapped around *netdev* (see Figure 2.1).

The `netdev_t` struct is the centerpiece of the API and contains as its members the driver itself, an event callback to handle events from the device, and a context variable that might be needed by the upper layers to identify themselves. The driver is a set of well-defined functions:

- `init()` is used to initialize a network device.
- `send()` is used to send a packet over the network device. It receives a list of (data, length) pairs as parameters to allow fragmented data to be sent over the device (so that e.g. headers and payloads can be stored in different places).
- `recv()` is used to receive a packet from the network device. It takes a buffer as parameter and returns the length of the packet currently in reception. If no buffer is given (i.e. the pointer to it is `NULL`), the length of the current packet is given, but in contrast to the case in which a buffer is given, the packet won't be marked as received. This can be used e.g. to allocate just enough memory before actually copying data into a network stack's buffer.
- `get()` is used to get an option value from the network device. Options are identified by the `netopt` data type. Additionally, the function receives a buffer to store the value. The function returns the value's length on success or a negative value on error or if the option is not supported by the device.
- `set()` to set an option value to the network device. `netopt` is also used here to identify an option and a buffer is given holding the desired value. On error,

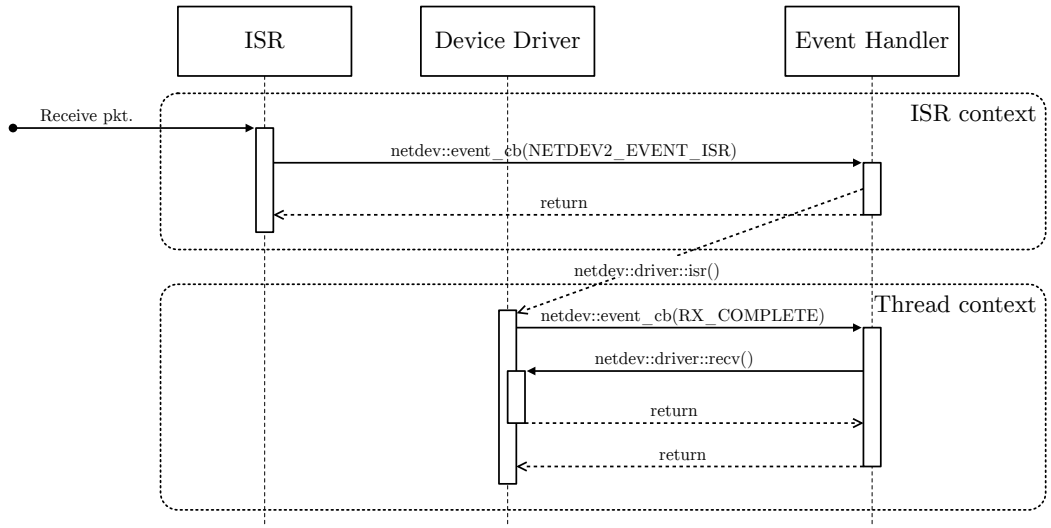


Figure 2.2: `netdev` event flow with `RX_COMPLETE` as an example

or if the device does not support the device, the function returns a negative error.

- `isr()` to tell the driver that an ISR context triggered an external event handler and now needs more information on the event.

As with all events in RIOT, an event to the network device is handled by the ISR (see subsection 2.2.2). However, we can't stay in the ISR too long so that other interrupts can be handled as fast as possible and to keep real-time capabilities. To achieve this, an external event handler can register an event callback to the *netdev*. The ISR handler of the network device driver uses this event callback to tell the external event handler that an event happened (see Figure 2.2 for a packet receive event). It is then up to the event handler to make a context switch from the ISR to a thread context. How this happens is out of scope of *netdev* itself, but using the IPC is an option. In the thread context it then uses the *isr()* function to notify the driver that we are indeed now in thread context and want more information on the event. The driver is then able to identify the event based on its own setup and notifies information on it back to the event handler using the event callback.

2.4 The transport layer connectiviy API – conn

`conn` is a collection of APIs to allow an application to access transport layer protocols network stack independent (see Figure 2.1). The main design goal was to offer a unified API for applications to implement against, independent of which network stack is used. Compared to its POSIX counterpart sockets, it is much more trimmed down and designed with the specific protocol in mind. The `conn` API for UDP `conn_udp` e.g. has no `connect()` call as a socket would have, since UDP is connection-

less. This allows the implementation of a `conn` for a stack and protocol to be very slim.

At the moment there are APIs defined for

- Raw IP (both IPv4 and IPv6) (`conn_ip`)
- TCP (over both IPv4 and IPv6) (`conn_tcp`)
- UDP (over both IPv4 and IPv6) (`conn_udp`)

All three APIs currently contain ways to both send and receive data and to set and get the addresses associated with a connectivity object. `conn_tcp` additionally supports functions to establish a connection.

2.5 Third-party Package System

RIOT is also equipped with the means to include third-party software – called packages (or `pkg` for short) – into a binary, not dissimilar to BSD ports [84]. Packages are comprised of a `Makefile` and a `Makefile.include` file that describe how to integrate the package into RIOT and a collection of patch files that describe how to change the third-party software to make it work with RIOT. Currently there is support for both Git- and HTTP-based sources, but since the integration is based on GNU Make [32], any other source type such as e.g. SVN is also possible.

For this thesis I provided a port for lwIP (see section 4.2) and emb6 (see section 4.3) using this system, but other software was ported as well. Among them are:

- *CCN-lite* [15], a network stack focused on Information-centric networking (ICN).
- *libcoap* [6], a high-level CoAP implementation providing wrappers around sockets to implement both CoAP clients and sockets.
- *microcoap* [79], a low-level CoAP implementation, providing capabilities to expose end-points for a CoAP server application.
- *OpenWSN* [86], a network stack which provides 6LoWPAN over IEEE 802.15.4e's Time-Slotted Channel Hopping (TSCH) MAC mode.

2.6 Summary

In this chapter we looked into the inner workings of the RIOT operating system and its network related APIs as well as its third-party package management.

The three main components of RIOT's real-time capable micro-kernel are its

- tick-less, preemptive scheduler,
- the IPC that allows for both non-blocking and blocking, and synchronous and asynchronous data communication between threads, and

- the mutual exclusion capabilities (mutex) that allow for synchronized access of global memory regions.

The common network device API `netdev` provides generalized access to all network devices supported by RIOT as well as way to bootstrap ISR-triggered events generated by the driver. This allows for easy integration of network devices into the RIOT ecosystem without ever touching non-device related code. To show-case the event-bootstrapping capabilities I described the use-case of receiving a packet.

On the other end of a network stack the transport layer connectivity API `conn` allows for access to any given network stack from an application. Contrary to POSIX sockets they don't provide a generalized API for all possible protocols, but provide a subset of functions for each protocol supported by a stack.

The third-party package management `pkg` of RIOT allows for easy integration of third-party software similar to BSD ports. Makefiles describe a way to both fetch the piece of software from a remote source and how to adapt them to run with RIOT.

3 The GNRC Network Stack

3.1 Introduction

GNRC (short for "generic") was developed in 2015 [69] as a replacement for the previous monolithic network stack[1] of the RIOT [3, 4, 88] operating system, with H. Petersen and myself as the main driving forces behind the effort.

I developed most of the IPv6 and 6LoWPAN parts of the stack, including NDP and 6Lo-ND.

Since it is part of RIOT, the stack is published under the LGPL version 2.1 and new functionalities and optimizations are currently in development by RIOT's worldwide community.

3.2 Design Objective

I already named a number of constraints in subsection 1.4.1 that *protocols* in the IoT are facing:

- *Large address space*
- *Low energy consumption*
  - *Low processing power*
  - *Memory constraints*
  - *Lossy communication medium*

When actually implementing a network stack, we also face the problem of the *high heterogeneity in embedded systems* in the IoT in many categories such as application space or hardware platforms, which I also mentioned in subsection 1.4.1. The traditional approach to deal with these problems is to provide multiple network stacks for a specific setup. This approach allows for very specific optimizations in that use-case. However, it naturally exacerbates the effort that needs to be put into implementation, testing, and other maintenance.

From its inception *GNRC* was designed to be both flexible enough for this heterogeneity but also efficient and small enough to hold the constraints we are facing in the IoT in general [69].

[1] It was very loosely based on uIP.

3.2.1 Requirements

Based on the foundations I stated above, I identified both functional and non-functional requirements, which I list in the following sections [69].

Functional Requirements

Focus on IPv6 With the big address space requirements and our focus on IETF protocols, focusing on IPv6 first was a natural decision. This way, we allow for both end-to-end connectivity between IoT devices and the *fringe Internet*[2] and are also able to make use of the IoT stack aimed at low-power and lossy networks, which I presented in section 1.4. Regardless, the stack should also be able to have an implementation of IPv4 or even more obscure network protocols later on.

Full-Featured Protocols that are implemented for *GNRC* should be implemented as completely as possible per their specification as a long-term goal, so the possibility to extend a module in the future must always be provided. This way we can attempt to find a generic solution that fits multiple use-cases instead of constraining the implementation by design.

Support for Multiple Network Interfaces The usual scenario in the IoT is a single node with one radio, but as seen in subsection 1.4.3, there is at least one border router needed per network. Usually, those are connected to the Internet via a wired connection such as Ethernet, but with the advent of new devices that support multiple radio standards [33], border routers between two different radio standards are also conceivable. For these reasons, support for multiple interfaces is required.

Parallel Data Handling Some embedded network stacks such as uIP [21] and emb6 [39] (see section 4.3), but also the predecessor of GNRC, try to save memory by reducing the packet buffer essentially to a single array that is able to store exactly one network packet at a time. While this might be suitable for some use-cases, it makes parallel data handling very hard. For scenarios with NDP [65] involved alone the protocol's implementation might need to queue packets for address resolution. Moreover, this often leads to massive data duplication as every protocol starts to establish its own local buffer, especially if heavy rework needs to be performed on a packet (e.g. with 6LoWPAN). While problems such as these might be solved by viewing them as special cases and copying packets around if need arises, I argue that designing the stack for parallel data handling from the ground up results in a cleaner and ultimately smaller code size.

[2] Term sometimes used for non-IoT Internet [75].

Non-Functional Requirements

Open Standards and Tools Since *GNRC* is part of the RIOT operating system it is of course – like the operating system itself – free software. I am convinced – as stated in section 1.1 – that this will ultimately lead to more secure and better maintained software [40]. Furthermore, the historic development of the Internet itself speaks for the fact that openness is the way to success: The most prevalent standards today such as IP or L2 technologies standardized by the Institute of Electrical and Electronics Engineers (IEEE) (e.g. WiFi or Bluetooth) are those that were specified openly or were released into the public domain later on. To achieve interoperability despite the heterogeneity of IoT devices, a network stack must adhere to this kind of standards. Besides, the development environment needs to be accessible and the software itself needs to be developed with open tools and well-known paradigms. Experiences with e.g. TinyOS's nesC language [56] show that exotic tools or paradigms can become a hindrance in reaching a significant number of developers to maintain a software in the long run.

Configurability While the stack needs to be adaptable to a number of use-cases, the set of configurable parameters should be held relatively small and some options should even aggregate from finer granulated options. However, having multiple degrees of freedom in the configurability means that understanding all of them will be harder for newcomers.

Modularity Besides providing good encapsulation – and thus less entangled code – a clean interface between modules also has the advantage of providing extensibility and easier testability for a module. The implementation of one component can, for instance, easily be switched out for another and thus provide different or extended functionality. But as with configurability, the modularity should rather be left on the coarse side in order to make the code more accessible to newcomers.

Low Memory Footprint We decided to not go for the lowest configuration possible, since we don't believe that supporting class 0 devices as defined by [8] would be possible with the given functional requirements. As such, class 1 devices ($\approx$ 10 KiB RAM, $\approx$ 100 KiB ROM) are our intended platform classes. Since one still wants to put application and other system code in the memory, we aimed for $\leq$ 30 KiB ROM and $\leq$ 10 KiB RAM[3] for a configuration with one interface, 6LoWPAN, IPv6, RPL, and UDP.

Low-Power Design IoT devices are expected to run for years on battery power as already mentioned in subsection 1.4.1, so low-power operation of all components of an operating system are required. Optimization for such operations have been shown

[3] Devices of this type have proven to have more RAM than just 10 KiB most of the time.

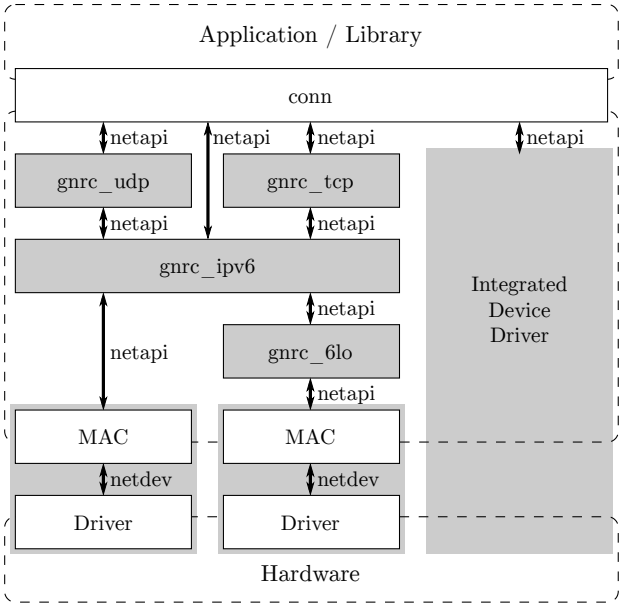


Figure 3.1: *GNRC* architecture overview

to be hard from experience, so it has to be included by design from the bottom up. This means I have to

a. allow for easy configuration of different protocol combinations in all layers down to the physical layer to adapt for given scenarios, and
b. the implementation and its data-structures must be suitable for maximum sleep intervals for the CPU.

3.3 Architecture

3.3.1 Overview

Based on the experience with the previous network stack, *GNRC* was designed not only to be highly modular, but also to have its modules be easily interchangeable. By this design, most of *GNRC*'s modularity can also be found in the code's file structure.

It is not dissimilar to UNIX STREAMS [71] in that it allows chaining of multiple modules behind one another through a common API. But while STREAMS uses input and output queues to exchange messages between the modules, *GNRC* utilizes RIOT's thread-targeted IPC: Every module – i.e. networking protocol – runs in its own thread which in turn has a message queue. The latter is optional in RIOT (see subsection 2.2.3) in general and even deactivated by default but is required for working with *GNRC*. The messages are in a well-defined format which is referred to as *"netapi"* in *GNRC*.

To determine which modules are interested in the packet next, a thread can look up potential receivers in a registry called *"netreg"* using the packet's type. Mod-

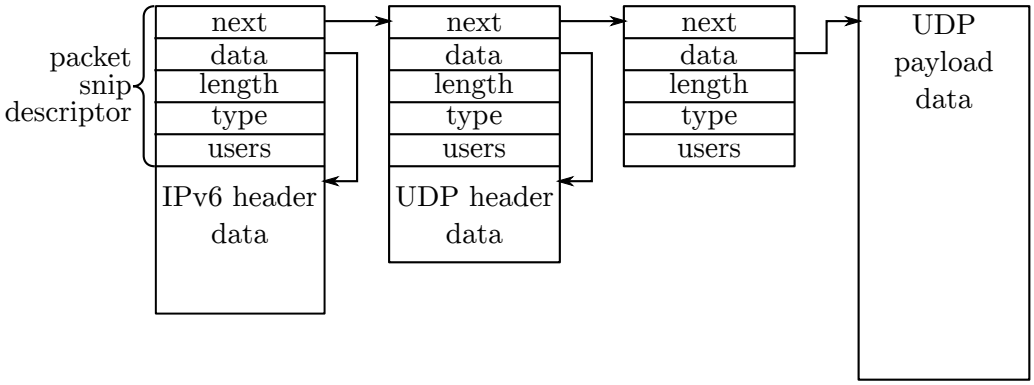


Figure 3.2: An example of a packet in transmission in the *GNRC* packet buffer

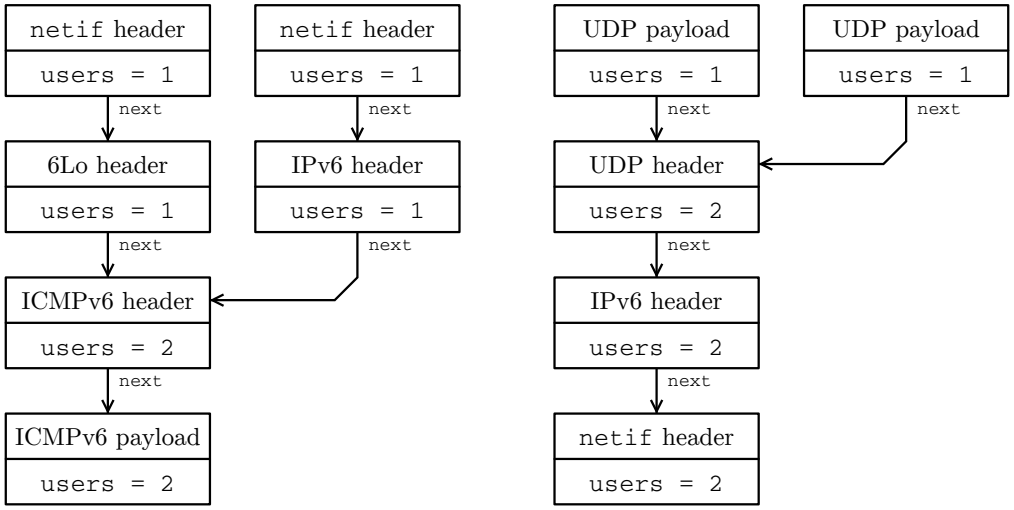
ules that are interested in a certain type can register themselves with this registry. Throughout the traversal time through the stack a packet is stored in *GNRC*'s packet buffer *pktbuf*.

With this architecture, *GNRC* achieves both the required modularity and an easy way to prioritize different parts of the stack – by assigning different priorities to the threads running the protocols and letting the operating system's process scheduler act upon these priorities. The trade-off is a slightly slower stack, but the IPC on RIOT is only one order of magnitude slower than direct function calls on typical IoT hardware [69, section 4.1], as the comparison with function-based stacks as lwIP (see section 4.2) or the event-queue polling emb6 stack (see section 4.3) in chapter 5 will show.

3.3.2 The Packet Buffer – pktbuf

Packets in GNRC are stored in a packet buffer called *pktbuf* with variable length chunks called "packet snips" (see Figure 3.2). They are usually used to differentiate between different headers and payloads. The data of the packets as well as the describing structures of the packet snips are stored in arrays allocated in static memory, but the API can in theory also handle dynamic allocation on the memory heap. Packet snips can be marked with a type to identify the content of the packet. Usually we refer to a list of packet snips in this context as packets, so the first member as the list's head is sometimes referred to synonymously as the packet. As such, the type of the first packet snip is usually referred to as the overall packet's type. To create a packet snip in *pktbuf* a user can use the function `gnrc_pktbuf_add()` which receives all user-controllable fields of a packet snip (`next`, `data`, `length`, and `type`) as arguments and returns the new snip.

Packet duplication throughout the stack is kept to a minimum, since data is copied or moved as little as possible on its way from network interface to application – ideally only once at each end-point of the stack – and vice-versa. As a result, the structures describing the packet (marked as the "packet snip descriptor" in



(a) An ICMPv6 packet in transmission being multicasted over both IEEE 802.15.4 and Ethernet

(b) A received UDP packet being handled by two applications

Figure 3.3: Examples of packets and their duplication in GNRC's packet buffer

Figure 3.2) can be stored independently of the actual data, so marking a packet header does not require moving the data around.

In order to be thread-safe, a packet snip keeps track of the threads using it via a reference counter (named `users` in Figure 3.2). The reference counter can be manipulated by the user with the `gnrc_pktbuf_hold()` and `gnrc_pktbuf_release()`. The first increments the counter, while the latter decrements it. When the counter reaches 0, the packet will be removed from the packet buffer.

If a thread wants to write to the packet and the reference counter is greater than one, the packet snip and all its next pointers will be duplicated *copy-on-write* style and the reference counter of the original snip is decremented by one. To keep duplication as low as possible, only pointers that can reach the current snip will be duplicated, giving a packet a tree structure in the packet buffer. For this reason, the order of the packet snips for a packet in reception is reversed – starting with the payload, ending with the header of the lowermost layer – while a packet in transmission is in the same order it will be transmitted in. This way, the number of pointers required to create such a tree structure is as low as possible. However, it needs to be pointed out that packets in reception will still keep their data in the received order for the most part. Only the packet snip descriptor list *marking out* the data is in reversed order. The reference counter of a parent will always be the sum of the reference counters of its child, because duplication also implies the decrementing of the packet snip's reference counter. This way, when a packet (i.e. the first snip in a packet snip list) is released from a thread, its own snips will be removed from the packet buffer, while the elements it shares with its copy are not (see Figure 3.3).

The functionality described in this paragraph is implemented in the `gnrc_pktbuf_start_write()` function.

If a received packet is handled by a protocol implementation it can *mark* its header using the function `gnrc_pktbuf_mark()`: A function that allows for the creation of a new packet snip representing the header. It has the length of the header and points to the original beginning of the old payload of the packet. The packet snip representing the payload is then updated to point to the offset of that header and the length of the header is subtracted from its length. It inherits the reference counter from the rest of the payload. To adhere to reception order, the payload is kept as a head of the packet snip list, while the new snip is inserted after the payload snip.

For sending a packet, a protocol expects a higher-layer protocol to prepend a header of the protocol's type to acquire addressing information like IP addresses or ports and no further parsing is required.

The advantages of storing packets this way is that we

1. are able to *handle multiple packets* at once, as the number of packets is not directly linked to the size of the buffer,
2. *save memory* by allowing the length of the packet to be dynamically allocated instead of using fixed-sized chunks that might not be used to their full size, and
3. *avoid duplication throughout the stack*, since it is a centralized buffer that every protocol implementation is able to use instead of keeping its own buffer (and save additional memory this way).

This way, I already allowed for two of our requirements for subsection 3.2.1 to be fulfilled: parallel data-handling and low-memory footprint. One could even argue that the avoidance of duplication saves energy, since the overall execution time is reduced.

Of course, we risk external fragmentation with this design, but I argue that due to the short lifetime of a packet in the buffer (only during its way through the stack) and the regularity in size of the transferred data, this effect will be negligible; experience from working with the packet buffer in real application settings confirm this argument.

The `GNRC_PKTBUF_SIZE` macro determines the buffer's size.

In summary, the operations provided by *pktbuf* are the following:

- `gnrc_pktbuf_add()` to allocated a packet in the packet buffer,
- `gnrc_pktbuf_hold()` to increment its reference counter,
- `gnrc_pktbuf_release()` to decrement its reference counter and to remove it from the packet buffer if the reference counter reaches zero,
- `gnrc_pktbuf_mark()` to mark a header in the packet, causing the creation of a new packet snip, and

- `gnrc_pktbuf_start_write()` to get privileged write access to a packet.

Additionally, some auxiliary functions not explicitly mentioned here are provided that help with data handling or conversion to other types. If they are mentioned in the remainder of the thesis, their functionality will be described there.

3.3.3 GNRC's Module Registry – netreg

At its core, the registry for *GNRC* (*netreg*) is a simple lookup-table that maps the type used to identify protocols as previously mentioned in subsection 3.3.2 to a list of PIDs (the receiving threads for the packet type). But an additional dimension (the "demultiplexing context") was added to allow threads the registration for certain protocol specific contexts (e.g. ports in UDP). `GNRC_NETREG_DEMUX_CTX_ALL` was added as a special context for all protocols, to allow for registration for packets of *all* contexts.

E.g. if a thread wants to send a message to a certain UDP port, it looks up the protocol type `GNRC_NETTYPE_UDP` with a demultiplexing context that equals that UDP port in the netreg using the function `gnrc_netreg_lookup()`. Every PID in the registry that meets both of these parameters would be a target for a message containing the UDP packet that has this port.

3.3.4 A Common Inter-Modular API – netapi

The *netapi* is the central API used by *GNRC* to communicate between the network layers that are registered to the *netreg*. It was designed to be as simple and versatile as possible so that even the most exotic network protocol can be implemented against it. I also wanted it to be as modular as possible to make it both easily extensible and testable. Its generic nature is what gave the *GNRC* network stack its name, as it could in fact be used independently from *GNRC*, e.g. to wrap a network stack implemented in hardware as it is provided e.g. with the CC3000 network device [83] (see Figure 3.1).

It is comprised of two asynchronous message types that do not expect a reply and two synchronous message types that are sent with `msg_send_receive()` (see subsection 2.2.3) and expect a reply in form of a message with the type `GNRC_NETAPI_MSG_TYPE_ACK`.

The two asynchronous message types are:

- `GNRC_NETAPI_MSG_TYPE_SND`, used to send packets (i.e. passing them "down" the stack). A pointer to a packet is expected to be in the data part of the message.
- `GNRC_NETAPI_MSG_TYPE_RCV`, used to receive packets (i.e. passing them "up" the stack). A pointer to a packet is expected to be in the data part of the message.

This is the only component of a *netapi* that is dependent on *GNRC* as it uses the `gnrc_pkt_t` struct of *pktbuf* in order to communicate the data. However, since for the communication itself only the data is important, it can be easily exchanged for more general structures such as e.g. POSIX's `struct iovec` to improve the genericness of *netapi*. However, for reasons of simplicity in the API's usage, I decided against this.

The synchronous message types are:

- `GNRC_NETAPI_MSG_TYPE_GET` to get an option value from a module. A `netopt` value to identify the option (see section 2.3) and a buffer to store the value are expected to be in the data part of the message. The `GNRC_NETAPI_MSG_TYPE_ACK` reply contains either the length of the value or a negative value to report an error or that the module does not support the requested option.
- `GNRC_NETAPI_MSG_TYPE_SET` to set an option value to a module. A `netopt` value to identify the option (see section 2.3) and a buffer with the value are expected to be in the data part of the message. The `GNRC_NETAPI_MSG_TYPE_ACK` reply contains either the length of the value set or a negative value to report an error or that the module does not support the requested option.

All of these message types are optional, but for the synchronous message types a value needs to be returned via the reply message that reflects that the option is not supported.

Note that for all of these messages, the semantics for *netapi* in general don't go beyond the ones specified. A network protocol that implements *netapi* however can require certain preconditions on packets or option values handed to it and implement behavior that goes beyond these specification. This way, the design goal to be as versatile as possible is guaranteed.

3.3.5 Network Interfaces

Network interfaces in *GNRC* are also represented as threads and use *netapi* for communication with the network layer. MAC protocols are implemented in the network interface's thread and access network device drivers directly through the *netdev* API (see section 2.3) since especially TDMA-based MAC protocols need access to the device with as little delay as possible.

As noted in the requirements, *GNRC* was written from the ground up to be a multi-interfaced, embedded network stack that supports both 6Lo- and non-6Lo-IPv6 communication natively just by providing multiple interface threads.

To keep the overhead for the stack itself low for handling L2 headers, every network interface converts the header format of their L2 protocol to a general interface header format – called *netif* header. This header contains the source and destination L2 addresses, their length and additional link metrics that can be utilized by the routing protocol, such as e.g. the LQI and RSSI. A common conversion API for popular L2 protocols assures that no duplicate porting efforts are required.

3.4 Description of Example Use-Cases

To showcase the interaction of the three main components of *GNRC* – *netapi*, *netreg*, and *pktbuf* – I present the comparatively complex use-cases of sending and receiving a UDP packet over 6LoWPAN (unfragmented). A graphical representation of these use-cases can be seen in the sequence diagrams in Figures 3.4 and 3.5, but since some operations take place module-internally, not everything that is happening is shown in that figure.

To save some space I use a special notation in both text and figures:

1. While many functions, variables, and constants in the actual code of *GNRC* (and in some of the text before) use a `gnrc_` (or capitalized `GNRC_`) prefix to associate it with the network stack's module, I omit this prefix in this example.
2. Capital letter words of protocol names in function calls denote the *protocol type* as it is used by *pktbuf* and *netreg* as described in sections 3.3.2 and 3.3.3.
3. `ALL` denotes the `GNRC_NETREG_DEMUX_CTX_ALL` demultiplexing context as described in subsection 3.3.3.
4. The tree of packet snips representing a packet is listed as a list of scalars and tuples `[X, (Y,Z)]`, where `X`, `Y`, and `Z` are *protocol types* as noted in 2. and a tuple representing a branching in the tree due to the *copy-on-write* mechanism described on page 23.
5. *netapi* messages are described with a shortened type notation and simplify `GNRC_NETAPI_MSG_TYPE_X` to `NETAPI_X`.
6. The data a *netapi* message carries in square brackets `[]` e.g. `NETAPI_SND[pkt =pkt]` denotes a `GNRC_NETAPI_MSG_TYPE_SND` message carrying a packet named `pkt`.

3.4.1 Reception of a UDP Over 6LoWPAN Packet

When a packet is received from an IEEE 802.15.4 device, it is handled with the mechanism already presented in Figure 2.2 to get out of ISR (see Figure 3.4). Note that *GNRC* uses the option of the `netdev::driver::recv()` method to use a `NULL` pointer to get the length of the packet first in order to allocate that length of data in the packet buffer using `pktbuf_add()`. Another call to `netdev::driver::recv()` with the allocated data buffer fills it with the received data. After determining the length of the IEEE 802.15.4 header we mark it in the data using `pktbuf_mark()` and replace it with a previously allocated (again by using `pktbuf_add()`) general L2 header – the *netif* header described in subsection 3.3.5. The remaining IEEE 802.15.4 header is released from the packet buffer using `pktbuf_release()`. The make-up of the packet snip list in `pkt` is now `[6LOWPAN, NETIF]`, with the `6LOWPAN` snip containing the whole 6LoWPAN datagram and the `NETIF` snip containing the *netif* header. Without 6LoWPAN (e.g. with an Ethernet frame of Ethertype IPv6) the packet snip list would look like this: `[IPV6, NETIF]`.

Since we are on an IEEE 802.15.4 interface we know to look for a 6LoWPAN handler in the *netreg* using `netreg_lookup()` (but other default protocols can be configured too or determined by a demultiplexing identifier like the Ethertype of Ethernet) with `ALL` demultiplexing contexts and receive the PID to the 6LoWPAN thread back. Here and in the following instances of `netreg_lookup()` I have to note that for simplicity's sake I only show a scalar PID being returned in Figure 3.4, but a list of PIDs is actually returned (although, since only the 6LoWPAN thread is registered in our scenario it would also just return a list of length 1). Since we only have one handler we don't actually need to call `pktbuf_hold()`, since we implicitly hand over any privileges to the receiving thread when we hand it over to the handler. Note however that we only leave it out in Figure 3.4 for all instances for simplicity's sake and to make it at least slightly easier to look at. In the code, this function is called in any case with an incremental value of 0 in the case of this scenario, since only one thread is a receiver of that message and the privileges to the packet are implicitly given up after sending the message to another thread. To hand the packet to the 6LoWPAN handler we hand it "up" the stack using a `NETAPI_RCV` message with the packet `pkt` as its content.

The 6LoWPAN handler then first write-protects the packet as described on page 23 using `pktbuf_start_write()` and, since we only have this thread referencing the packet, receives exactly the same packet back. If the reference counter of the packet had been greater than 1 due to a `pktbuf_hold()` call, it would have received a copy of `pkt`.

The 6LoWPAN handler determines that the packet contains a compressed IPv6 and UDP header and allocates space for the uncompressed instances in the packet buffer using `pktbuf_add()`. Since we are *receiving a packet* the IPv6 header is appended to the UDP header.

After decompressing the compressed headers we mark them using `pktbuf_mark()` (since the length of them depends greatly on their contents, we have to do this afterwards) and replace them with the already allocated IPv6 and UDP headers. We then release the remaining 6LoWPAN dispatch from the packet buffer and also set the payload type to `UNDEF` so UDP can find it later. The resulting packet now looks like this: `[UNDEF,UDP,IPV6,NETIF]`.

Again we use `netreg_lookup()` to find all handlers for `IPV6` for the `ALL` demultiplexing context and use `pktbuf_hold()` to increment the reference counter for potential receivers other than the first and send it to the IPv6 header using `NETAPI_RCV`.

Once more we assure in the IPv6 handler that we can write safely using `pktbuf_start_write()`, but in terms of packet parsing, the IPv6 handler does not have to do much, since 6LoWPAN already provided the packet snips that IPv6 would normally mark. If 6LoWPAN wasn't present (e.g. for an Ethernet interface), the IPv6 handler would have to mark the IPv6 header itself using `pktbuf_mark()` here. It then uses the next header field in the IPv6 header to identify the next

handler, which is of course UDP, looks it up for the ALL context in the *netreg* accordingly, and sends it using NETAPI_RCV. Because we have a 6LoWPAN use-case, the packet make-up does not change in IPv6 in this use-case and is still [UNDEF,UDP, IPV6,NETIF]. Without 6LoWPAN the type of the payload would be changed to the type of the payload – UDP – and the packet would look like this: [UDP,IPV6,NETIF].

The UDP handler also uses pktbuf_start_write() to potentially receive a write-safe copy of the packet. As with IPv6, the 6LoWPAN handler already created a UDP snip for the header, so no further parsing is needed. But again, for the non-6LoWPAN case, this would have happened here. As such the packet looks still the same: [UNDEF,UDP,IPV6,NETIF]; without 6LoWPAN, we would finally reach the same make-up, too.

Finally, we look up the PIDs for the applications interested in packets with the destination port of the UDP packets using netreg_lookup() with the UDP type and the destination port as demultiplexing context, hold the packet if needed for those handlers (again only if the number is greater than 1) and hand it over using NETAPI_RCV.

3.4.2 Transmission of a UDP Over 6LoWPAN Packet

UDP packets are transmitted with *GNRC* using NETAPI_SND from an application or library (like *GNRC*'s conn implementation e.g.). UDP expects both an IPV6 (or any network layer protocol header) and UDP packet snip present in addition to the packet snip representing the payload. So right from the start, the packet has the make-up [IPV6,UDP,UNDEF]. Note that the order isn't reversed like in the transmission case, as described in subsection 3.3.2. Additionally UDP expects at least the destination port field of the UDP header set, as was also described in subsection 3.3.2. As with reception, we first make sure we are the only ones having write access to the packet using pktbuf_start_write() (see Figure 3.5). For simplicity, we assume the packet was not sent somewhere else and that the packet's reference counter is 1, so the packet isn't duplicated and just returned.

UDP then sets the unset field in the UDP header – typically source port and length – however, the check-sum will be calculated later, since we don't know all IPv6 addresses yet, needed for the pseudo-header [19]. Next, it identifies that the network header is of type IPV6 and searches for its handler in the *netreg* using the ALL demultiplexing context, finds the IPv6 thread (again – here and below – for brevity we only show the scalar ipv6_pid in Figure 3.5, so no pktbuf_hold() is necessary), an send it to that thread using NETAPI_SND with the packet as data.

The IPv6 thread again uses pktbuf_start_write() to get write-access to the packet and sets the remaining IPv6 header fields, using e.g. internal functions to determine the source address from the destination address, if it isn't set already. This is also the point were the check-sum of upper layers – UDP in this case – is calculated, using a callback back to the module of the upper layer. After that, it uses next-hop determination to determine the interface the packet is supposed to be sent

over and uses address resolution to determine the destination L2 address. The IPv6 thread writes both of these informations into a `netif` header (see subsection 3.3.5) that it creates using `pktbuf_add()`, and prepends it to the packet. The make-up of the packet is now `[NETIF,IPV6,UDP,UNDEF]`.

Since the interface we are sending over is a 6LoWPAN interface (the IPv6 thread identifies this through a flag associated to the interface) the IPv6 thread looks up the 6LoWPAN handler using `netreg_lookup()` with the `ALL` context and sends the packet to it using `NETAPI_SND`. For a non-6LoWPAN interface, the packet would just have sent the packet directly to the interface using `NETAPI_SND` to the PID identifying the interface and the procedures below concerning 6LoWPAN are skipped.

6LoWPAN needs to identify the maximum length of frames the L2 can send. It uses `NETAPI_GET` with parameter `MAX_PKT_LEN` to get this from the interface. It knows over which interface to send the packet, thanks to the packet's `netif` header which contains this information. The interface's MAC thread in turn uses `netdev::driver ::get()` with the same parameter to get the value from the device driver.

Afterwards, the 6LoWPAN thread gains write access to the packet using `pktbuf_start_write()`. It allocates a packet snip with enough space to write 6LoWPAN compression data into it using `pktbuf_add()` and prepends it to the payload. It then compresses the IPv6 and UDP header and writes the compression data into the newly allocated snip. During this process, the actual length of the compression data can be determined and the handler is thus able to resize the packet using `pktbuf_realloc_data()`. Since they are no longer needed, the IPv6 header and UDP header are released.

Using the interface identifier in the `netif` header, the 6LoWPAN thread then sends the packet to the interface's thread using `NETAPI_SND`.

The interface uses *pktbuf*'s function `pktbuf_get_iovec()` to convert the list of packet snips into an array of POSIX **`struct`** `iovec` that is then used to send the data over the network device using `netdev::driver::send()`.

3.5 Protocol Support

As one of the requirements for the development of *GNRC* was to focus on IPv6 first, the protocols outlined by the IoT stack (see section 1.4) are fully supported. This means *GNRC* IPv6, 6LoWPAN, UDP, and RPL. However, I decided to omit some sub-features normally found in well-known operating systems and network stacks for now, to be added later. Among them are the duplicate address detection as outlined by RFC 4862 [85] as well as the multi-hop duplicate address detection outlined in RFC 6775 [76] and the HC1 header compression from RFC 4944 [64], as it was made obsolete in RFC 6282 [41]. As was required, all features provided are optional.

Though UDP is already implemented and well-tested, TCP support is currently still in development [13]. There are no plans for IPv4 support as of yet.

Through both its `conn` layer and the POSIX sockets provided by RIOT, *GNRC* can support CoAP through any CoAP library and RIOT already provides support for both `libcoap` [6] and `microcoap` [79] through its third-party package system (see section 2.5).

3.6 Summary

This chapter described the *GNRC* network stack – the centerpiece of this thesis. I drafted out the short history of this stack and made our design objective clear – including *GNRC*'s requirements, both functional and non-functional:

- The functional requirements were
  - focus on IPv6,
  - full-featured in protocol support,
  - support for multiple interfaces, and
  - parallel data handling.
- The non-functional requirements were
  - use of open standards and tools,
  - easy configurability,
  - modularity,
  - low memory footprint, and
  - low-power design.

I then described the overall architecture of *GNRC* and went into detail for all of its three main components, which are

- the central packet buffer *pktbuf*,
- the network module registry *netreg*, and
- the inter-module communication API *netapi*,

In contrast to prior work, the intermodular (and thus in turn inter-layer and inter-protocol) communication between the components of the *GNRC* stack is based on multi-threading IPC, rather than function- or event-queue-based approaches: Each protocol is running in its own thread and uses the IPC-based API *netapi* to communicate with others. Easy prioritization of protocols is possible this way by simply using the priorities of the threads.

To show the interaction between the parts of *GNRC*, I drew out the two use-cases of receiving and sending a UDP packet over 6LoWPAN, but also made the differences clear when no 6LoWPAN is used in the sub-use-cases where this was important.

Finally, I made clear where *GNRC* is currently at regarding protocol support, as it currently only supports UDP over IPv6 (+ 6LoWPAN), with application layer protocol support being able to be imported using `conn`.

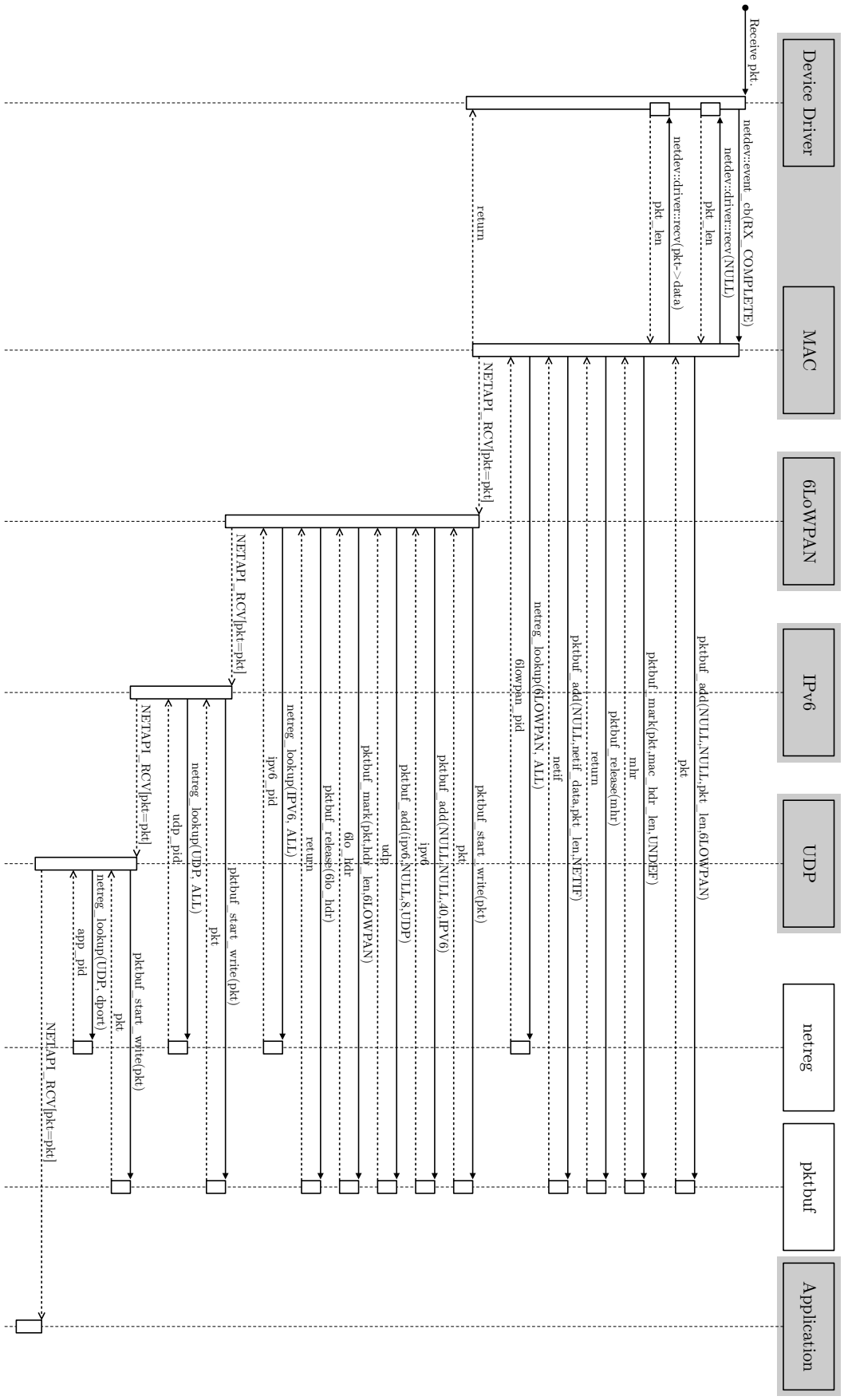


Figure 3.4: Example use-case for GNRC: Reception of a UDP packet over 6LoWPAN. Gray boxes show that the module is running in its own thread.

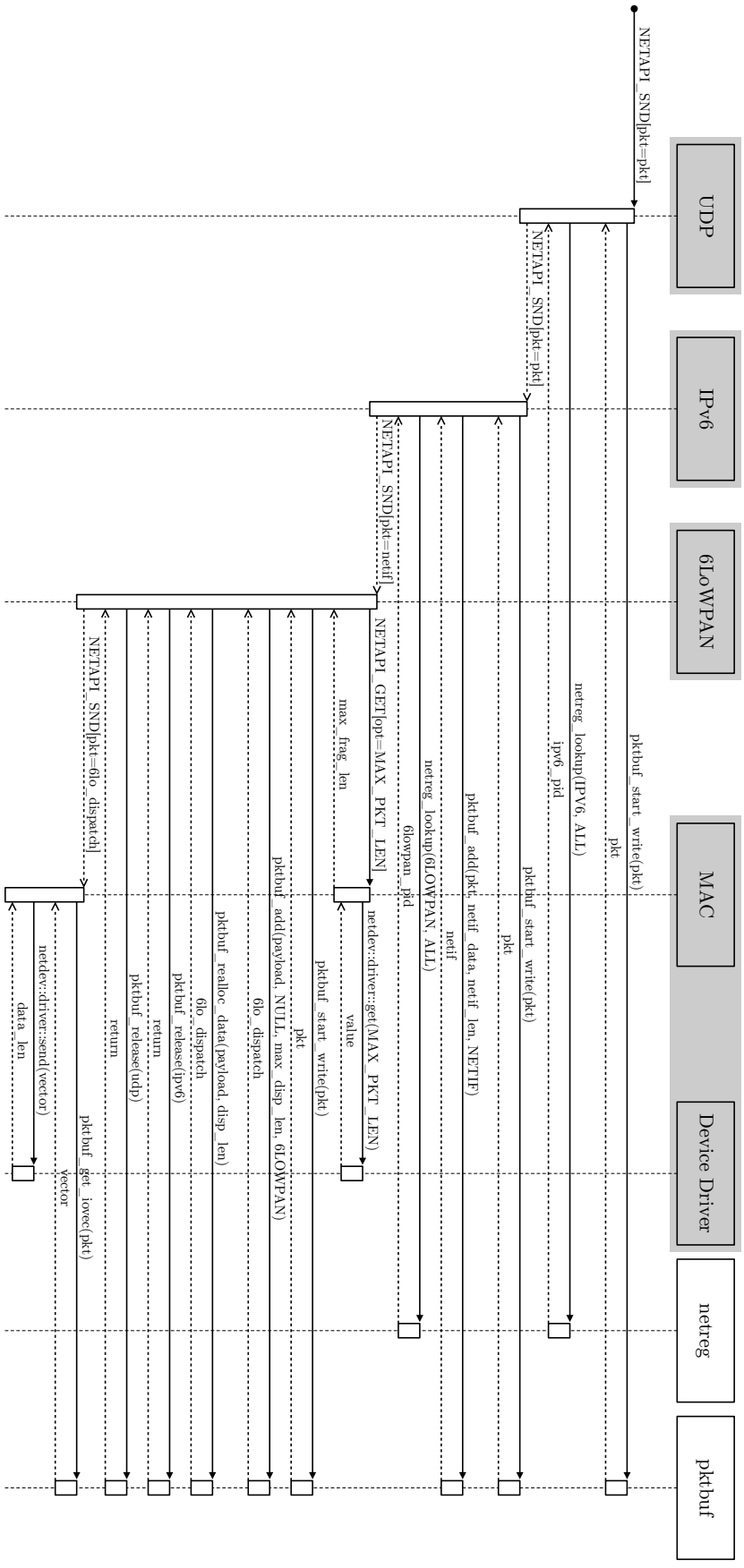


Figure 3.5: Example use-case for GNRC: Transmission of a UDP packet over 6LoWPAN. Gray boxes show that the module is running in its own thread.

4 Other Network Stacks

4.1 Introduction

To get an idea of the performance of GNRC, I decided to compare the stack to two well-established solutions. However, many network stacks are tightly integrated into their operating system's environment, which makes it hard to compare them just on the basis of their own performance.

Two stacks that are independent of any operating systems proved to be of particular interest. *lwIP* [20] – which is often used in combination with FreeRTOS [5] – is very easy to port thanks to its OS abstraction and Hardware Abstraction Layer (HAL). *emb6* – a fork of *lwIP*'s sister stack *uIP* – on the other hand was developed to run stand-alone on many boards that RIOT already supports. As such, it was very easy to plug into *emb6*'s HAL.

Both stacks were ported to RIOT utilizing its third-party package management system (see section 2.5).

The following sections broadly describe the inner workings of the two stacks. Specifically, section 4.2 is dedicated to *lwIP*, while section 4.3 goes into *emb6*.

4.2 lwIP

The *lwIP* (lightweight IP) network stack was originally developed in 2001 by A. Dunkels at the Swedish Institute of Computer Science (SICS) [20]. Development today is continued by a worldwide community under a modified 3-clause BSD license [31].

Modularity Like GNRC, *lwIP* was developed to be highly modular. This modularity is achieved by setting `LWIP_<MODULENAME>` macros to a non-zero value. The default values for this can be found in `src/include/lwip/opt.h` and can be modified in an external `lwipopts.h`. The C pre-processor then includes or leaves out the relevant functionality according to the setting of these macros.

Stack-internal Communication To hand a packet from layer to layer, *lwIP* uses a centralized IPC message queue that the `tcpip` thread reads (see Figure 4.1). Messages can be identified by a type and contain the callback needed for the protocol type that is then called by the `tcpip` thread.

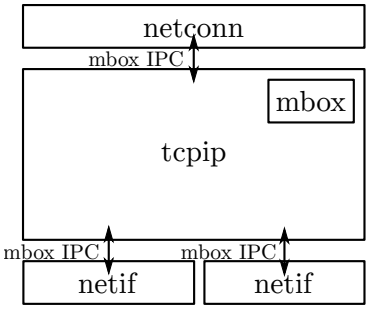


Figure 4.1: lwIP architecture overview

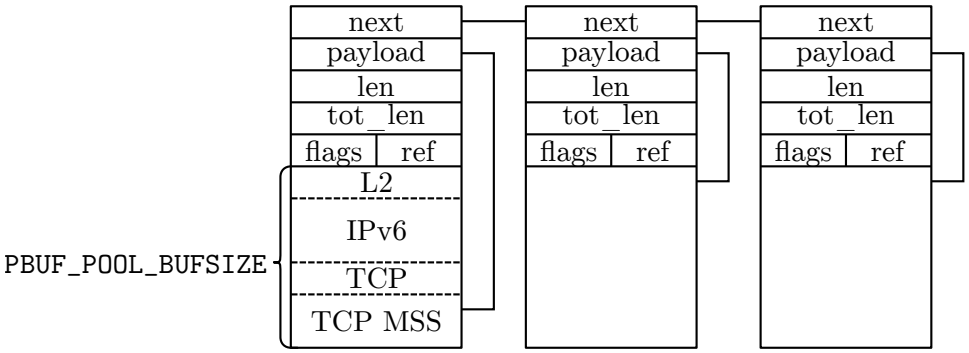


Figure 4.2: lwip packet buffer concept in PBUF_POOL mode [20, section 6.1]

Packet Buffering For packet buffering, *lwIP* uses a central packet buffer `pbuf` that allocates fixed-sized chunks in memory. The size of each chunk in PBUF_POOL mode (which is the default and which I will use for my experiments) can be modified with the PBUF_POOL_BUFSIZE macro, which is set to a predefined value that can accommodate a TCP segment of minimum segment size plus IPv6 and link layer headers. If a packet exceeds the size of a chunk, it concatenates it with additional chunks as needed (see Figure 4.2). The chunks and list structure for the packet buffer can be either allocated dynamically in the memory heap or in static memory in a memory pool. For the latter, the pool size can be set with the MEM_SIZE macro.

Network Interfaces *lwIP* supports multiple interfaces. These are implemented in an object-oriented manner as a list of structures, containing option values and an `input()` method to receive a packet allocated and marked for further handling in the packet buffer and up to three output methods:

- `output()` to send IPv4 packets over the interface,
- `output_ip6()` to send IPv6 packets over the interface, and
- `linkoutput()` for both `output()` and `output_ip6()`, but also for modules like the Address Resolution Protocol (ARP) implementation of *lwIP* to hand an already assembled L2 frame to the actual network device.

Protocol Support *lwIP* was originally developed with focus to have a full TCP implementation on embedded devices, but it also provided UDP functionality. Though it only supported IPv4 [20] in the beginning, IPv6 was later added as well. As of February 22nd, 2016, a 6LoWPAN layer is also provided as a `netif` adaptation layer [94] (by setting `input()` and `output_ip6()` of an interface to the according 6LoWPAN functions). There is however no full deployment of 6Lo-ND according to RFC 6775 [76], nor is there an implementation for RPL or even static routing yet. Other protocols supported are IPv4, ARP, Point-to-Point Protocol (PPP) and application layer protocols such as for instance the Domain Name System (DNS). As *lwIP* is modular, most of the supported features are of course optional.

Since *lwIP* has full socket support, any CoAP library based on sockets or able to utilize sockets can be used to bring CoAP to *lwIP*.

4.3 emb6

emb6 was forked early in 2015 from Dunkel's second network stack and the primary network stack of the Contiki operating system – uIP ("micro IP") [21] – in an effort by the working group of A. Sikora et al. at Hochschule Offenburg to provide a uIP version without the need for Contiki's proto-threads [39]. This was achieved primarily by replacing the message queues of Contiki with similarly structured polling queues and refactoring some of Contiki's functionalities and data structures into emb6's `utils` sub-module.

As with Contiki and uIP, *emb6* was released under the 3-clause BSD license.

Modularity Since it is based on uIP, it is similarly monolithic as uIP. There are a few features that can be deactivated in order to reduce code size and the code itself is highly structured, but the borders between functionalities are not as clear-cut as in *lwIP* and *GNRC*. For instance, most of the network stack's internal functionality is implemented in the `emb6/src/net/ipv6` directory, including UDP and TCP.

Stack-internal Communication *emb6* is designed to run in a single thread and is even able to just be called in a periodic manner so other tasks can be handled in-between. Normally, however, it polls the event queue on a fixed time interval and acts upon an event if it finds one. As such, it is not only purely function-based, but also – since it mostly operates on global variables and buffers – lacks parameters for those functions for the most part.

Packet Buffering Buffering in *emb6* (and uIP) is primarily done by the global `uip_buf` array. It fits exactly one IPv6 packet (or less if need be) and is used for both sending and reception of IP traffic. Its size can be configured with the `UIP_CONF_BUFFER_SIZE` macro. Packets of higher layers are assembled in-place in this

buffer. 6LoWPAN (or `sicslowpan` as it is called here due to its SICS legacy) uses its own buffer for packet compression, fragmentation and reassembly. For *emb6* their size is determined by the `QUEUEBUF_CONF_NUM` and `QUEUEBUF_CONF_REF_NUM` macros.

Network Interfaces *emb6* only has support for one network interface, though a SLIP interface can be added to allow for IPv6 communication via the device's UART interface.

Protocol Support Since it only uses a single buffer and does not implement sliding window, *emb6* inherits the property of uIP of only being able to handle single-segmented TCP streams [21]. uIP's IPv4 support was dropped from emb6, most likely to save space and maintenance effort. But apart from this, 6LoWPAN, IPv6, RPL, and UDP are supported, although as with lwIP (see section 4.2) there is no full support for 6Lo-ND.

Due to its uIP/Contiki legacy, *emb6* also has native CoAP support through the Erbium [47] library.

4.4 Summary

In this chapter we looked into the two other stacks we used for the evaluation of GNRC: *lwIP* and *emb6*.

On the one hand we have *lwIP* which is very modular and versatile, and on the other we have *emb6* which inherits (and even exacerbates) the monolithic nature of *uIP*.

Both stacks are capable of running without a given operating system and are thus perfect to be put into context with *GNRC* in the common environment of the RIOT operating system.

5 Comparative Evaluation of GNRC, lwIP and emb6

5.1 Introduction

In chapter 3 I described the GNRC network stack that I developed for this thesis and presented two further stacks for comparison in chapter 4. In this chapter I intend to compare those three both qualitatively (see section 5.2) and quantitatively via experimentation (see section 5.3). I decided to mainly focus on stack traversal times (see subsection 5.3.2) and memory consumption (see subsection 5.3.3), since they provided me with the most meaning given the tools I had at my disposal. With the results of both comparisons, I will then discuss the pros and cons of my new network stack in section 5.4.

5.2 Qualitative Comparison

An overview over the supported features of each stack we are about to compare can be seen in Table 5.2. Due to lwIP being the oldest and not stripped down for a certain use-case as emb6 is, it has the richest feature set. It basically has full IPv6 and TCP support. Features not listed are its IPv4 (including ARP), PPP, and application layer protocols such as for instance DNS (see section 4.2), which allow lwIP to also be used in more classical applications outside the IoT use-case. As described in section 4.2, 6LoWPAN support in lwIP is comparably young [94]. For this reason, some features such as fragment re-sequencing, old-style

	6LoWPAN							ICMPv6											
		Frag.												RPL					
Stack	**multi-iface.**	**reseq.**	**mult.**	**HC1**	**IPHC**	**NHC**	**IPv6**	**error**	**echo**	**NDP**	**SLAAC**	**6Lo-ND**	**MLD**	**st.**	**non-st.**	**TCP**	**UDP**	**CoAP**
GNRC	✓	✓	✓	✗	✓	✓	✓	●	✓	✓	✗	✓	✗	✓	●	✗	✓	✦
lwIP	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✦
emb6	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	●	✓	✓

Table 5.2: Comparison of network stack features (✓ = supported, ✗ = not supported, ● = partially supported, ✦ = support through external library)

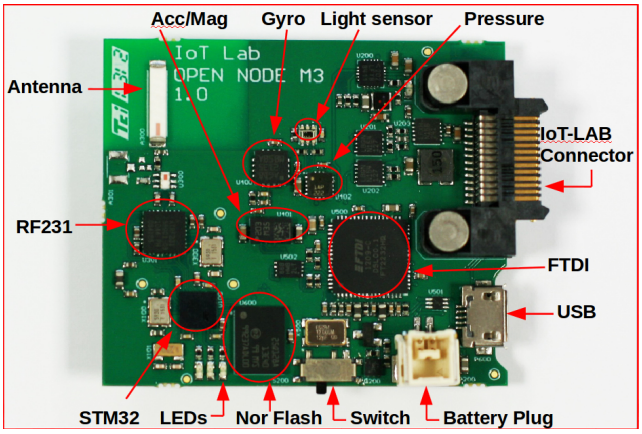


Figure 5.1: The FIT IoT-LAB Open Node M3 [29]

header compression, and optimization for neighbor discovery aren't available yet. Likewise, RPL is not implemented for the stack yet. The two more IoT-focused stacks emb6 and GNRC, on the other hand, support a large subset of the features of 6LoWPAN; emb6 even has partial TCP support (see section 4.3). However, due to its limitations, emb6 is only able to receive one 6LoWPAN fragment at a time and doesn't optimize for neighbor discovery yet. GNRC, on the other hand, doesn't have any TCP support yet (see section 3.5). However, capability for some features that aren't established yet are already provided, namely ICMPv6 error reporting and RPL non-storing mode; for both features, the foundations are given, but they are not yet used. For ICMPv6 error reporting, the actual sending of the messages is still missing, while the reporting API already exists; for RPL non-storing mode, support for source routes is given both by the IPv6 extension header support and the forwarding tables capabilities of RIOT.

Code quality-wise, I of course found myself most comfortable around GNRC since I wrote most of its code and documentation. For the other two stacks, I was for the most part of the impression that the documentation only served as an explanation for the implementation. As such, a lot of code inspection was required to actually find out what each component was doing. But as I can't exactly say how the experience of a new user to GNRC would be (I heard both praise and criticism from the community), I can't know if this is the case for GNRC as well. All-in-all I view this aspect as very subjective and would therefore prefer to refrain from any further actual comparison.

5.3 Comparison via Experimentation

5.3.1 Experimentation Platform

All of my experiments were run on the M3 platform of the FIT IoT-LAB testbed [30]. It is equipped with an ARM Cortex M3 processor (ST2M32F103REY, 72 MHz,

64 KiB RAM) and 2.4 GHz 802.15.4 radio (AT86RF231) and a number of sensors (see Figure 5.1) [29]. The testbed is openly available and only requires registration to the FIT IoT-LAB infrastructure.

I only require one of these nodes for the experiments presented in this section, so in order to obtain consistency in the output of the experiments I chose a fixed node at the Paris site of the testbed: `m3-43`.

To get the same type of behavior concerning context changes and timings, I ported all stacks to the RIOT operating system – except for GNRC, which is already native to RIOT. Apart from providing a system access layer for both emb6 and lwIP, it was also necessary to wrap their network interface layer around `netdev` (see section 2.3). Since I used UDP for all my experiments, I implemented the `conn_udp` sub-module of the `conn` API (see section 2.4) of for each stack.

I used commit `20ba062` [55] in the RIOT repository for all my experiments. At the time of experimentation, this was the current `HEAD` of the 2016.04 release branch. The experiments were based on `9d4cbb2` of my application repository [53].

I used version `4.9.3 20150529` of the GCC ARM Embedded [90] toolchain on Ubuntu 16.04 to build all my experiments.

Unless otherwise stated, I used the default configuration provided by RIOT.

5.3.2 Setup For Traversal Time Tests

To get a benchmark for the stacks, I split the experimentation into transmission of packets and their reception. To have a fair and equal baseline for the comparison I only included protocols that every stack supports; as such, I chose UDP as the transport layer protocol, IPv6 for the network layer with 6LoWPAN with fragmentation, IPHC and NHC, and used no special MAC on the IEEE 802.15.4-based L2. To access UDP, I used the `conn_udp` API provided in the RIOT ports. Instead of using a physical device, I developed a virtual `netdev_test` device driver that allows us to switch out its method functions to e.g. stop or start a timer.

There are a few constants that are used for both experiment setups. The four most significant are `EXP_MIN_PAYLOAD`, `EXP_MAX_PAYLOAD`, `EXP_PAYLOAD_STEP`, and `EXP_RUNS`.

`EXP_MIN_PAYLOAD` is the minimum payload length we want to send. It was set to 8, because for emb6 and lwIP, 0 wasn't a possible payload length and I wanted to choose a value divisible by 8, since 6LoWPAN only fragments in intervals of length divisible by 8 [64, section 5.3]. Additionally, this gives me the change in the transmission experiments to add a payload of at least size 5 to help us identify a packet in a simple manner for timing of UDP transmission stack traversal times.

`EXP_MAX_PAYLOAD` is the maximum payload length we want to send. It was set to 1232, because 1280 bytes is the MTU for 6LoWPAN [64, section 4]: Since this also includes the IPv6 header itself (40 bytes) and the UDP header (8 bytes), we subtract their length from 1280 bytes and obtain 1232 bytes as the result.

EXP_PAYLOAD_STEP is the number of bytes by which we want to increment the payload length after one payload length was finished testing. For our purposes we set this value to 8, again to have the payload always be divisible by 8.

EXP_RUNS is the number of times we send packets of a certain payload length through the stack to later calculate the mean and standard deviation of all runs for the payload.

Stack Configuration

I also optimized the packet buffer sizes of the stacks to allow for exactly one 1232 bytes payload to be handleable by the respective stack in both the transmission and the reception direction. I did this to both allow for the fairest comparison for size, as described in subsection 5.3.3, and to limit parallel data handling capabilities as much as possible (creating stack-internal congestion artificially). The values were determined by both approximating a range of values by careful analysis and then finding the exact value through trial-and-error.

All stacks were configured as IPv6 routers with full 6LoWPAN support (as far as the stack could provide it) and to only be able to handle UDP.

GNRC only needed its packet buffer resized to GNRC_PKTBUF_SIZE = 1676, to fit one IPv6 datagram + one fragment + meta-data.

lwIP required the following configuration macros to be set:

- PBUF_POOL_BUFSIZE = 200, to fit one fragment + meta-data into one packet buffer chunk. lwIP allocates a chunk per fragment on reassembly.
- MEM_SIZE = THREAD_STACKSIZE_DEFAULT[1] + 3624, to fit the memory stack of the tcpip thread, a full IPv6 packet, 6LoWPAN reassembly structures and additional data allocated by the network stack.

Furthermore, I deactivated Multicast Listener Discovery (MLD) and IPv6 fragmentation and reassembly, since the other two stacks don't support these features, and set the temporary buffer of my netdev layer implementation for lwIP LWIP_NETDEV2_BUFLEN to 127 (the length of an IEEE 802.15.4 frame).

emb6 requires the following macros to be set:

- UIP_CONF_BUFFER_SIZE = 1332 to fit a full IPv6 datagram + additional data
- QUEUEBUF_CONF_NUM = 16 and QUEUEBUF_CONF_REF_NUM = 16 to allow for reassembly of 1240 bytes sized IPv6 datagrams.

[1] Provided by RIOT's thread.h.

The polling delay for the emb6 thread was determined experimentally to be 58 μs. This way, the stack traversal times will be as short as possible while still being mostly stable for payload lengths.

UDP Transmission

The essential parts of the experimentation code can be found in Listing 5.1.

After we setup the virtual device to provide the stack with the option values it requires and initialized the stack, we add the `_netdev_send()` callback to the device so that we are able to stop the traversal time (l. 8).

For emb6 and GNRC I tested both with and without RPL. lwIP is exempt from these, since it neither provides a RPL implementation, nor does it allow for setting of static routes, as already noted in section 4.2. For experiments without RPL, I used a link-local (`fe80::/64` prefixed [38]) address as destination address `dst`. With RPL, I used a `abcd::/64` prefixed global unicast address as `dst` and set up the node as a RPL root with an address of the same prefix as the DODAG ID.

After further experiment-related setup of the network stack, we start sending our UDP packets and steadily increment the overall payload length by `EXP_PAYLOAD_STEP` from `EXP_MIN_PAYLOAD` until we reach `EXP_MAX_PAYLOAD` (ll. 11-30).

We use the last 8 bit of the experiment run counter as ID for the timer and fill the payload with that value, but leave 4 bytes (encoded in `TAIL_LEN`) empty (l. 16). Afterwards, we append both the current payload length (l. 19) and an identifying sequence I dubbed *"honey guide"* for the purpose of these experiments (l. 21) to the payload to get to the full payload length. The honey guide is a constant 2 byte sequence that was generated randomly at compile time and is used to discern the packets sent by this experiment from other packets sent by the stack in a simple manner.

We then start our timer, store the time in a global array (l. 24), and send the packet through the stack (l. 26).

On the `netdev` side of the stack we immediately stop the timer (l. 37) and try to identify the packet. Note that the IPv6 datagram might have been fragmented by 6LoWPAN, so only the last fragment will carry the honey guide. This is why we also appended the payload length in l. 19, since the size of the fragment might of course be smaller than that. Fragments that aren't the last or other packets that the stack sends out periodically or on initialization are therefore filtered out by identifying those that do not carry the honey guide sequence. After identifying the packet we acquire the start time of our timer from the global times array (l. 54) and print out the pair of payload length and traversal time (l. 56) for data collection.

Listing 5.1: UDP transmission experiment code

```
/* includes and global variable defintions ... */
static int _netdev_send(netdev_t *dev, const struct iovec *vector,
                        int count);

void exp_run(void)
{
    /* register send callback to `netdev_test` device */
    netdev_test_set_send_cb(&netdevs[0], _netdev_send);
    /* setup `dst` and configure stack to be able to send to `dst` (add
     * `dst` to neighbor cache + set default route if `dst` is global) */
    for (payload_size = EXP_MIN_PAYLOAD;
         payload_size <= EXP_MAX_PAYLOAD;
         payload_size += EXP_PAYLOAD_STEP) {
        for (unsigned id = 0; id < EXP_RUNS; id++) {
            /* fill payload with ID for timer stop */
            memset(buffer, id & 0xff, (payload_size - TAIL_LEN));
            /* append experiment marker sequence
             * (length + random, but fixed 16 bit) */
            memcpy((uint16_t *)&buffer[payload_size - TAIL_LEN],
                   &payload_size, sizeof(uint16_t));
            memcpy(&buffer[payload_size - HONEYGUIDE_LEN],
                   honeyguide, sizeof(honeyguide));
            /* start traversal timer */
            timer_window[id % TIMER_WINDOW_SIZE] = xtimer_now();
            /* send UDP packet to `dst` */
            conn_udp_sendto(buffer, payload_size, &unspec,
                            sizeof(unspec), &dst, sizeof(dst), AF_INET6,
                            EXP_SRC_PORT, EXP_DST_PORT);
        }
    }
}

static int _netdev_send(netdev_t *dev, const struct iovec *vector,
                        int count)
{
    /* stop traversal timer */
    const uint32_t stop = xtimer_now();
    /* get data from packet vector */
    const uint8_t *payload = vector[count - 1].iov_base;
    const size_t payload_len = vector[count - 1].iov_len;
    int res = 0;
    uint32_t start;
    uint16_t exp_payload_len;
    uint8_t id;

    /* filter out unwanted packets (e.g. by NDP or RPL or non-terminal
     * 6LoWPAN fragments) by comparing last to bytes with `honeyguide` */

    /* get payload length of packet and timer ID from packet */
    memcpy(&exp_payload_len, &payload[payload_len - TAIL_LEN],
           sizeof(uint16_t));
    id = payload[payload_len - TAIL_LEN - 1];
    /* get start of traversal time */
    start = timer_window[id % TIMER_WINDOW_SIZE];
    /* output result for current packet */
    printf("%u,%" PRIu32 "\n", (unsigned)exp_payload_len,
           (stop - start));
    return res;
}
```

UDP Reception

The essential parts of the experimentation code can be found in Listing 5.2.

This time, however, we setup the receive callback of the `netdev_test` device for stopping the traversal time (l. 8).

As with the transmission experiments, we test for both emb6 and GNRC with RPL and without RPL. Though the behavior of the stack isn't affected this time because we don't perform look-ups in the FIB, it might have some influence on the timings of the stack since it is sending out RPL-related data. Therefore, we add a `abcd::/64` prefixed global unicast address to the interface of the stack we are using and setup the node as a RPL root with the same address as DODAG ID. After further experiment-related setup of the network stack we start a thread for UDP packet reception (l. 14). It will loop over a `conn_udp_recvfrom()` and stop the reception times (function at l. 72, in-depth description below).

We then start to prepare and trigger the reception of 6LoWPAN packets that might be fragmented depending on the payload length and increment the overall payload length by `EXP_PAYLOAD_STEP` from `EXP_MIN_PAYLOAD` until we reach `EXP_MAX_PAYLOAD` (ll. 20-46).

To simplify the start of the traversal timer in the `netdev`, we also store the current `_id` globally (l. 25).

Then, we prepare and count the required fragments for the current payload length (l. 28). For brevity, I included only a comment in Listing 5.2 at this point, but actually we only set the payload to `id & 0xff` here and recalculate the UDP checksum. Most of the preparation on the headers already happened before the loop over `id` even started (marked by the comment in l. 22). Note that even in experiments with this, fragments will be addressed with link-local addresses (or rather speaking with full elision of the addresses in 6LoWPAN header compression [41]). RPL is only included to account for timing differences due to the intermediate sending of additional RPL traffic.

For every fragment, we then trigger a reception event at the `netdev` (l. 38) but wait using a semaphore for the data of the fragment to be copied into the stack's in order to not accidentally overwrite the fragments buffer on the next run (l. 42).

The `netdev` uses the global variable `_frag` to keep track of which fragment it is currently at. Thus, after all fragments have been handled, we need to reset that variable to 0 (l. 44).

The receive callback of the `netdev` first checks if the `buf` parameter is a `NULL` pointer (l. 51). This is required since it is a legal value in the `netdev` API to cause the device to just return the length of the current frame in the buffer (which we do in l. 69).

If `buf` isn't `NULL`, we start the stack traversal timer (l. 58) with the help of the global variable `_id` which we updated in l. 25 and copy the current fragment into `buf` (l. 61).

Afterwards, we signal the main thread waiting at the semaphore `sync` in `exp_run()` (at l. 42) that the data was copied and return the length of the current fragment to the caller (l. 66).

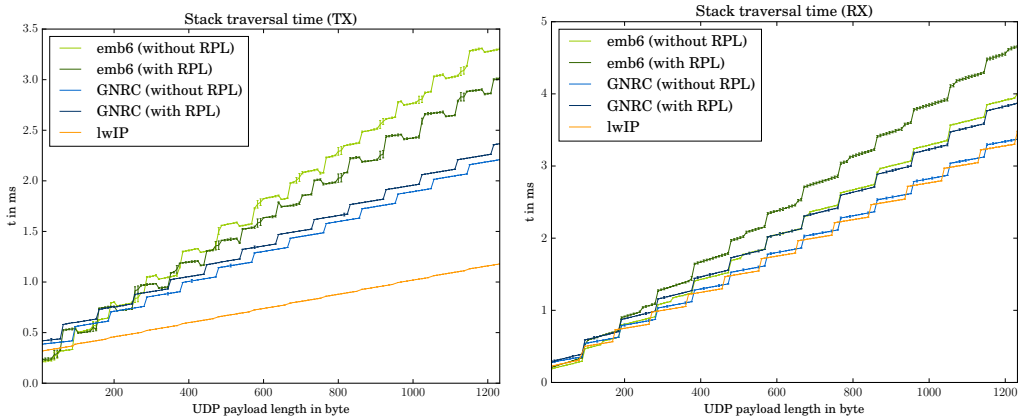
During all of the above, the receiver thread created a UDP connection (l. 82) and started listening for an incoming UDP packet (l. 86). As soon as a packet is received, the transmission timer is stopped (l. 89) and the first byte of the UDP payload is used to identify the packet (l. 91). We do not need to filter or make use of any special "honey guides" here, since we only receive the packets that were generated above.

Finally, we print out the resulting payload length and traversal time pair (l. 93).

Listing 5.2: UDP reception experiment code

```
1  /* includes and global variable definitions ... */
2  static int _netdev_recv(netdev_t *dev, char *buf, int len, void *info);
3  static void *_recv_thread(void *arg);
4
5  void exp_run(void)
6  {
7      /* register receive callback to `netdev_test` device */
8      netdev_test_set_recv_cb(&netdevs[0], _netdev_recv);
9
10     /* setup `dst` and configure stack to be able to send to `dst`
11      * (add `dst` to interface if need be) */
12
13     /* start packet reception thread */
14     if (thread_create(thread_stack, sizeof(thread_stack), THREAD_PRIO,
15                       THREAD_CREATE_STACKTEST, _recv_thread, NULL,
16                       "exp_receiver") < 0) {
17         return;
18     }
19
20     for (payload_size = EXP_MIN_PAYLOAD; payload_size <= EXP_MAX_PAYLOAD;
21          payload_size += EXP_PAYLOAD_STEP) {
22         /* prepare 6LoWPAN headers according to payload_size */
23         for (unsigned id = 0; id < EXP_RUNS; id++) {
24             /* store ID globally */
25             _id = id;
26             unsigned fragments = 0;
27
28             /* prepare and count fragments (or just 1 datagram if
29              * `payload_size` is small enough) according to
30              * `payload_size` in `frag_buf[i]` and fill UDP
31              * payload with (`id` & 0xff);
32              * store the lengths of fragment `i` in `frag_buf_len[i]` */
33
34             for (unsigned i = 0; i < fragments; i++) {
35                 /* trigger receive event in `netdev` for fragment i */
36                 netdev->event_callback((netdev_t *)netdev,
37                                        NETDEV2_EVENT_ISR,
38                                        netdev->isr_arg);
```

```
39                  /* wait for `netdev` to copy data of fragment `i`
40                   * into stack to not override in next iteration
41                   * before it was copied */
42                  sema_wait(&sync);
43              }
44              _frag = 0;  /* reset fragment counter for `netdev` */
45          }
46      }
47  }
48
49  static int _netdev_recv(netdev_t *dev, char *buf, int len, void *info)
50  {
51      if (buf != NULL) {
52          uint8_t frag;
53          /* error handling etc. */
54          /* get current fragment prepared in `exp_run()` */
55          frag = _frag++;
56          if (frag == 0) {
57              /* start timer on first fragment */
58              timer_window[_id % TIMER_WINDOW_SIZE] = xtimer_now();
59          }
60          /* copy current fragment into stack buffer */
61          memcpy(buf, frag_buf[frag], frag_buf_len[frag]);
62          /* signal main thread (`exp_run()`) that fragment was copied
63           * to stack buffer */
64          sema_post(&sync);
65          /* return length of current fragment (not `_frag` anymore) */
66          return frag_buf_len[frag];
67      }
68      /* return length of current fragment for buffer preparation */
69      return frag_buf_len[_frag];
70  }
71
72  static void *_recv_thread(void *arg)
73  {
74      conn_udp_t conn;
75      ipv6_addr_t addr = IPV6_ADDR_UNSPECIFIED;
76      size_t addr_len;
77      uint16_t port;
78      /* initialize thread's message queue */
79      msg_init_queue(thread_msg_queue, THREAD_MSG_QUEUE_SIZE);
80      /* create connection object */
81      memset(&conn, 0, sizeof(conn));
82      conn_udp_create(&conn, &addr, sizeof(addr), AF_INET6, EXP_DST_PORT);
83      while (1) {
84          int res;
85          /* wait for incoming packet */
86          if ((res = conn_udp_recvfrom(&conn, recv_buffer, sizeof(
                recv_buffer),
87                                       &addr, &addr_len, &port)) > 0) {
88              /* stop timer */
89              uint32_t stop = xtimer_now();
90              /* take ID for timer from packet */
91              uint8_t id = recv_buffer[0];
92              /* output result for current packet */
93              printf("%d,%" PRIu32 "\n", res,
94                     stop - timer_window[id % TIMER_WINDOW_SIZE]);
95          }
96      }
97      return NULL;
98  }
```



(a) UDP transmission (b) UDP reception

Figure 5.2: Stack traversal times by payload

5.3.3 Setup For Memory Consumption Measurements

To test the size of static memory, I use the same application for UDP reception timing presented in section 5.3.2, but I added the remaining functions of the `udp_conn` (`gnrc_conn_close()`, `gnrc_conn_sendto()`, and `gnrc_conn_getlocaladdr()`) at reachable points of the code to not have them and the functionality behind them optimized out by the compiler when it optimizes for size.

To check the stack usage I modified both applications from subsection 5.3.2 to print the sum of both stack usage and stack size for each thread instead of the timings. The stack usage can be acquired with RIOT's API function `thread_measure_stack_free()` when the `DEVELHELP` macro is set and the thread in question has the `THREAD_CREATE_STACKTEST`[2] flag set. I excluded the `idle`, `main` and `exp_receiver` (created in Listing 5.2, l. 14) thread from those sums. The maximum of all of these runs of all experiments for each of the remaining memory stack shall be my result.

5.3.4 Discussion of the Results

Stack Traversal Time

The results for the stack traversal times can be seen in Figure 5.2. The lines represent the means of all `EXP_RUNS` for one payload length, while the error bars represent the standard deviation of these.

The significant steps in the graphs are results of the 6LoWPAN fragmentation. In the tests without RPL, I used IPv6 link-local addresses based on the IEEE 802.15.4 long address, so the addresses are fully compressed. For the tests with RPL, I used the `abcd::/64` prefix (without any context given), which means that the prefix is carried inline and the packets are 8 bytes longer. As a result, the fragmentation

[2]This flag is usually set for the stacks of most threads on their creation.

for transmission for the tests without RPL starts at 96 bytes payload length, while the fragmentation with RPL starts at 88 bytes payload length, since the headers need 8 bytes more space. For both cases, it then continues to add further fragments for every 96 bytes of payload length. For reception, I did not use the prefix in 6LoWPAN's compression header, so the transition of payload lengths generating new fragments is the same for both tests with RPL and without.

For lwIP, the fragmentation also starts at 88 bytes due to a known bug [54] in the version of lwIP used that considers the already compressed IPv6 datagram for its datagram length and offset fields instead of the uncompressed datagram [41, section 2]. Due to the 8 byte unit system used for the offset field, this gives the payload less space in the first fragment resulting in the 88 byte fragment. The 6LoWPAN frame preparations in the reception tests needed adoptions to mitigate this issue.

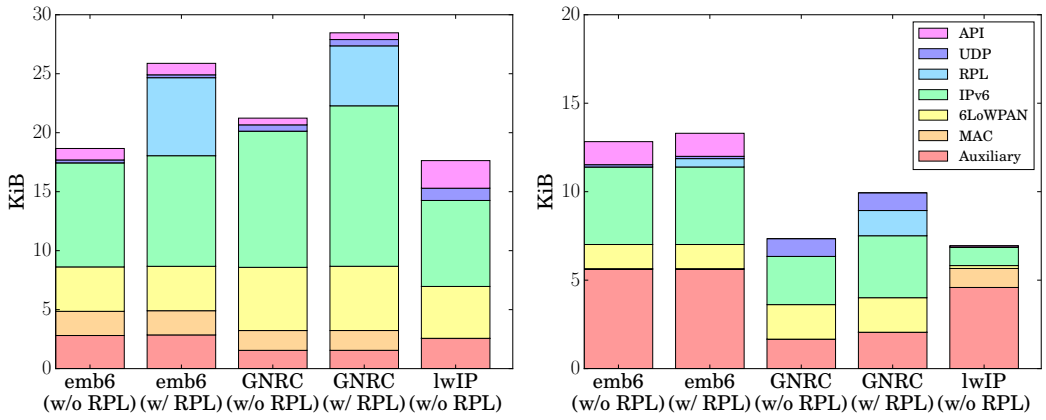
Though lwIP's transmission stack traversal time seems quite linear in the graphs, it also shows the steps in the data, though not very steeply. While the other stacks gain ≈ 100 μs for each fragment they must send, lwIP only gains ≈ 20 μs. GNRC has this disadvantage since its 6LoWPAN thread yields between each fragment to prefer received frames, while emb6's disadvantage is most likely due to the slower packet queue used for fragmentation. A more thorough code analysis is needed to explain the huge time differences regarding fragmentation between emb6 and lwIP while using similar techniques to send a datagram.

emb6's irregularities in the steps and bigger deviation from the mean can be explained by its sleep-cycle-based event polling. If an event was added to the event queue while the emb6 thread was sleeping, the event can be handled immediately, while an empty queue results in another 58 μs (see section 5.3.2) for the stack to wait.

Though both lwIP and GNRC use thread-based communication for passing messages, lwIP is slightly faster (especially – as mentioned – in the transmission). lwIP uses an `mbox`-based IPC messaging that is – at the current state of RIOT – much slower than the thread-targeted IPC messaging used by GNRC[3]. Apart from the initial `conn` to `tcpip` thread, however, this mechanism isn't used for transmission, which again explains the much slower growth in transition time. GNRC on the other hand uses multiple IPC calls between its layers and is still faster than the event and sleep-cycle-based emb6. This only proves the point I made in subsection 3.3.1, that GNRC is not at a disadvantage – at least on RIOT – despite its excessive IPC-usage. With a proper `mbox` IPC – as it was recently added to RIOT [73] – lwIP might even be faster.

While this fast message handling in transition is impressive, it is important to note that this is achieved by effectively blocking the `tcpip` thread with the sending task. In an application setting with short requests and long replies – as is usually

[3]This is because the centralized 'mbox'-based IPC implementation used in these experiments actually uses the thread-targeted IPC messaging in the background.



(a) ROM size of the stacks (b) RAM size of the stacks

Figure 5.3: Size of the network stacks in memory on iotlab-m3

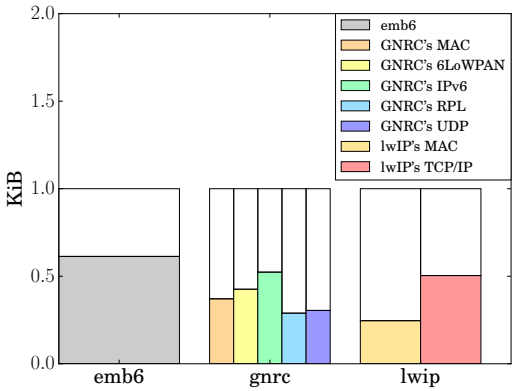


Figure 5.4: Memory stack usage of the 1 KiB (`THREAD_STACKSIZE_DEFAULT`) allocated for them on Cortex-M platforms by network stack and thread

the case with CoAP for instance – this might lead to incoming requests getting lost since the `mbox` is filled, because 6LoWPAN is still sending. As mentioned above, GNRC on the other hand lets the 6LoWPAN thread yield between every fragment, giving the usually higher prioritized network interface thread the opportunity to handle a potentially received packet. The 6LoWPAN thread then either continues to send out fragments or handles received packets depending on which command is at the head of the IPC queue. Send responses can't get lost this way either, since they only are solicited when a received packet is handled by the thread responsible for them.

RPL of course slows down both emb6 and GNRC minimally, since both stacks need to send out RPL packets in-between the data packets.

Memory Consumption

For the analysis of the build size, I used the `cosy` tool by H. Petersen [68]. Apart from graphically splitting up the memory usage of each compile unit, it is able

to print out the build tree, allowing for a precise attribution of each byte to the corresponding part of the code down to the function level. With a parsing script I was able to assign each part of the flash image to the part of the stack it provides the functionality for. The results of this can be seen in Figure 5.3. "Auxiliary" denotes everything not related to a certain layer, like packet buffers, memory management, OS wrapper functions, or event handling.

For memory stack usage (see Figure 5.4) I used the modified applications I described in subsection 5.3.3 and printed out the stack usage instead of timings (which can be obtained with the `DEVELHELP` compile flag activated).

As mentioned in section 5.3.2, I configured the stacks to transmit and receive exactly one 6LoWPAN-MTU-sized IPv6 datagram with UDP payload. As such, with its separate buffers for 6LoWPAN and `uip_buf`, for the rest of the stack it uses the most RAM, while in its default configuration it will most likely be the smallest.

The fact that RAM-wise GNRC is bigger than lwIP is not surprising. Since it uses a full default-sized memory stack for each of its layers in addition to its packet buffer, there is a lot more memory needed, but as the analysis of the stack usage in Figure 5.4 shows: the memory stacks for both GNRC and lwIP can be reduced to at least $\approx 60\%$ (some even more than 50%) of their size. However, this means that GNRC uses about 2–2.5 KiB (or 2.5-3 KiB with RPL) less, while lwIP only would take about 1.5 KiB less. emb6 on the other hand only uses one thread and also needs about 60% of its memory stack. As follows, we can only reduce the stack size by about 0.4 KiB ($\approx$ 400 bytes). Yet it is important to note that this is only true for this particular scenario of the test. A more realistic usage might require more memory stack size for any of the network stacks.

In ROM usage, both emb6 (through uIP) and lwIP show the maturity of their code base. However, GNRC is only about 3-4 KiB bigger than emb6 and about 4-5 KiB bigger than lwIP, which is a code size that could be reduced in the future by thorough optimization. GNRC's code is considerably younger than the implementation of the other two ($\approx$15 years of lwIP/uIP vs. $\approx$1 year of GNRC) and still has much room for optimization. Especially its IPv6 and 6LoWPAN implementation could most likely be decreased to a size of lwIP's counterparts, since they are the biggest parts of any stack, but as in lwIP, these two layers are very isolated from their surroundings in GNRC. This means that these optimizations could be carried out internally since their current code size seems not to be bound to the overall architecture of GNRC.

The memory size requirements set in subsection 3.2.1 of $\leq$ 10 KiB RAM and $\leq$ 30 KiB ROM I managed to keep for GNRC regardless.

5.3.5 Overall Discussion

Overall, lwIP seems to be the most suitable stack under the parameters and conditions I tested, with GNRC being a close second.

lwIP has the most mature feature set at the network layer level, though its 6LoWPAN port is quite young and it is still missing a working RPL implementation.

It is the smallest stack, both RAM- and ROM-wise, although both GNRC and emb6 can be configured to reach the same RAM sizes with only small drawbacks such as constraints on the size of handleable packets in emb6's case or by careful analysis of the overall usage of memory stack and packet buffer in GNRC's case.

Especially lwIP's fast stack traversal time for packet transmission needs to be pointed out, though it acquires this by effectively blocking the reception of packets that might be received during 6LoWPAN fragmentation.

Both GNRC and lwIP have a very predictable and smooth stack traversal time, while emb6's sleep-cycle-based polling of the event queue does not guarantee the same timings for packets of the same length which also might be bad for the real-time context of RIOT. In comparison, Contiki's uIP, which emb6 is based on, might not have this problem and might actually have a speed comparable to GNRC, since it uses Contiki's proto-threads for the event management, allowing the stack to only become active if and when an event was triggered (i.e. a packet was sent or received).

5.4 The State of GNRC

From this evaluation and both my own as well as my colleagues' experience working with the stack, I managed to isolate a few advantages and disadvantages of GNRC both compared to the stacks used in this comparison and in general.

Advantages

- The well-defined interface enforces clear communication between single network stack modules and encourages deployment of new protocol implementations.
- Also resulting from the clean interface, sending of user-generated data (or data generated in the application layer) is very easy to trace and their path throughout the stack is comprehensible and simple to describe as a use-case (as show-cased in section 3.4).
- The usage of IPC for the API allows per design parallel data handling of multiple network messages.
- The loose coupling between networking modules given by the IPC interface allows for both
  - straight forward and simple composition of network stack modules, and
  - re-composition and re-compilation of such configuration even during runtime – hot-plug style. This allows for very simple update capabilies where only parts of the stack can be taken down and restarted later.

- Due to its variable-size paradigm, the packet buffer is simple to configure: There is just one degree of freedom – its overall size. It can thus be shrunken down very easily to the most constrained memory-size feasible for the use-case (e.g. only allow for handling of one 6LoWPAN fragment at a time). For packet buffers using fixed-size chunks, as for example the one used in my experiments for lwIP, multiple parameters need to be finely tuned to achieve something like this, while the choice of values isn't always that transparent.

Disadvantages

- IPC is hard to debug. While most of the user-generated traffic is straight forward and easy to describe stack-internally, it becomes harder when automated messages and especially timed messages come into play (as they are for instance generated by stack-internal control mechanisms like neighbor discovery or routing). Due to the way RIOT's IPC works, a packets call history basically gets lost at each module border. For those cases an organized method of sending IPC messages needs to be found.
- Due to the multi-threaded nature of the stack each module needs its own memory stack. Since at development time it is hard to estimate the actual usage of the stack, lots of allocated memory actually stays unused for the most part.
- While in theory a network stack is cleanly layered, this strict view is rather academic. In practice, neighboring layers sometimes need knowledge of their neighboring protocols and even protocols further up or down the stack. UDP for example needs knowledge of the network protocol and which addresses it will use to calculate its check sum and IPv6 has to have at least an idea what L2 a packet is sent over. When it comes to routing, which is usually set above the network layer, link-layer information is sometimes needed for routing metrics. In a stack where each layer runs in its own thread (or multiple) ways to communicate this information needs to be found.

5.5 Summary

This chapter evaluates GNRC both in comparison with two other stacks – lwIP and emb6 – and on its own terms. For comparison, I used the feature-rich and mature lwIP stack and emb6, which is more of a stripped-down stack for special use-cases as a base line. Traversal times in transmission and reception and their memory consumption were the factors that were compared qualitatively. The mature lwIP stack proved to be the best of the three, with my GNRC stack being a close second. Future optimization work will go into mitigating this advantage of lwIP.

In general, it was shown that GNRC's heavy reliance on its IPC-based API can be seen as both a blessing and a curse: It gives the stack a very clean and structured

architecture that provides many of the requirements just by being built upon IPC, but also causes problems in the development process.

All in all, GNRC is able to hold up compared to the established solutions, both performance-wise and in keeping the constraints it faces.

6 Conclusion & Outlook

6.1 Conclusion

This thesis documented the design and development of the GNRC network stack for the RIOT operating system. I described the domain of the IETF IoT protocol suite the network stack is supposed to work with and the RIOT operating system and its relevant components it was implemented for. Afterwards, I described the GNRC network stack and its three main components – *netapi*, *netreg*, and *pktbuf* – in detail. For the sake of comparison, I also described the lwIP and emb6 network stacks briefly. For the evaluation, I not only compared the three stacks based on their feature set, but also on the basis of experimentation with UDP over 6LoWPAN traffic. Here, I assessed both network stack traversal times and memory consumption of the stacks. lwIP was proven to be the best of the three stacks, with GNRC being a close second. I also determined general advantages and disadvantages of GNRC.

Although lwIP proved to be the better stack with regards to my experiments, I still argue that GNRC is the better solution for RIOT specifically. Disregarding arguments like optimization efforts to be done, GNRC was after all developed with real-time in mind. Both the property of lwIP's 6LoWPAN implementation of potentially blocking reception when sending out larger data frames and the heavy reliance on `malloc()`-like memory allocation make it less suitable for real-time applications.

Likewise, all stacks can be stripped down in size with a trade-off in functionality, so that lwIP's size advantage is only as apparent as it is when considering the scenario that allowed for a fair comparison between the stacks.

Therefore, GNRC remains the perfect candidate for a network stack in RIOT as a real-time operating system for the IoT.

6.2 Outlook

As the youngest of the three stacks compared here, GNRC still has the potential to be more efficient, especially since its performance is very close to lwIP's in almost all categories.

Further experimentation needs to be carried out; especially in-depth power consumption tests could be of further interest. Other testing parameters, such as for example packet throughput and real-world scenarios with actual radio transmissions, might also be of interest for future research in this area. Especially the performance

under high load should be interesting with the sending behavior of fragmented packets of emb6 and lwIP compared to the more reception-lenient way in which GNRC handles fragmentation. Furthermore, there are a few stacks that were not considered, such as the network stack of the operating system TinyOS [57] or the original uIP stack of Contiki, as well as its RIME network stack [22]. OpenWSN [86] and even comparisons to non-IP stacks like CCN-lite are also conceivable. Both are already ported to RIOT (see section 2.5).

However, there must be a very clear consideration of the problem space, especially for CCN-lite and OpenWSN which differ in their function set from the purely 6LoWPAN-based stacks.

Further development on GNRC and an expansion of its feature set, which is a given with RIOT's rise in popularity, will also increase the necessity of a future repetition of the comparisons provided in this thesis.

On the actual development work side of things a few issues are open. The disadvantages stated in section 5.4 need to be mitigated or at least addressed.

For the debugging problems, a code of conduct in the usage of popular debugging tools like `gdb` might already be very helpful, but also some kind of debugging and back-tracing capabilities for IPC messages could come in handy. This is however a broader topic that does not only concerns the network stack, but IPC in general. Future research needs to be put into this topic to find out if and how a solution could be provided.

On the problems at hand regarding the ability to debug in GNRC, it turns out that most of the problems stem from a very incomprehensive implementation of the NDP and not due to the design of the network stack itself: Due to time constraints during the implementation, I was forced to develop this part of GNRC in a very ad-hoc manner and without an overarching plan. A common network layer API, like the one presented by Ee et al. [25], might provide a better way to track packets generated by the network protocols themselves and help find the root cause of some major issues.

For the memory consumption of the stack we already hinted at the possibility of optimization. More experimentation based on real-world scenarios needs to be done to find a good default value for each memory stack on each platform. Another possibility might be to merge some protocols that often come together for certain use-cases. Viable candidates for this are for example 6LoWPAN and IPv6.

The disadvantage of cross-layer requirements were already addressed for the most part: L2 metrics are provided through the `netif` header and certain assertions (e.g. a packet sent over UDP already needs its IP layer allocated before entering the UDP module) are made regarding the structure of a packet through-out the stack. For the checksum calculation, callbacks were introduced that are called as soon all values needed for the checksum are set. I find reassurance in the fact that we were always able to come up with comprehensible and resource-friendly solutions for these

requirements, that future cross-layer requirements will solicit similar kinds of solutions.

Bibliography

[1] Arduino. Arduino. https://arduino.cc, 2016. [Online] Accessed: 2016-06-19.

[2] ARM Ltd. mbed OS. https://mbed.org/technology/os/, 2016. [Online] Accessed: 2016-06-02.

[3] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[4] E. Baccelli, O. Hahm, M. Wählisch, M. Günes, and T. Schmidt. RIOT: One OS to Rule Them All in the IoT. Technical report, Freie Universität Berlin and INRIA, 2012.

[5] R. Barry. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[6] O. Bergmann. libcoap: C-implementation of coap. https://libcoap.net, 2016. [Online] Accessed: 2016-05-22.

[7] C. Borchert, D. Lohmann, and O. Spinczyk. CiAO/IP: A Highly Configurable Aspect-oriented IP Stack. In *Proc. of ACM MobiSys*, pages 435–448. ACM, 2012.

[8] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228 (Informational), 2014. http://tools.ietf.org/html/rfc7228.

[9] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), 2013. http://tools.ietf.org/html/rfc7049.

[10] A. Brandt and J. Buron. Transmission of IPv6 Packets over ITU-T G.9959 Networks. RFC 7428 (Proposed Standard), 2015. http://tools.ietf.org/html/rfc7428.

[11] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), 2014. http://tools.ietf.org/html/rfc7159.

[12] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C Recommendation*, 2008.

[13] S. Brummer. gnrc tcp. https://github.com/RIOT-OS/RIOT/pull/4744/, 2016. [Online] Accessed: 2016-06-19.

[14] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference On*, pages 233–244. IEEE, 2008.

[15] CCN-lite. CCN-lite: Lightweight implementation of the Content Centric Networking protocol. `http://ccn-lite.net/`, 2015. [Online] Accessed: 2016-06-19.

[16] L. Coetzee and J. Eksteen. The Internet of Things-promise for the future? An introduction. In *IST-Africa Conference Proceedings, 2011*, pages 1–9. IEEE, 2011.

[17] E. da Silva Santos, M. Vieira, and L. Vieira. Routing IPv6 over wireless networks with low-memory devices. In *Proc. of IEEE PIMRC*, pages 2398–2402. IEEE, 2013.

[18] S. Davidson. Open-source hardware. *IEEE design & test of computers*, 21(5):456, 2004.

[19] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), 1998. `https://tools.ietf.org/html/rfc2460`.

[20] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. `http://static2.wikia.nocookie.net/__cb20100724070440/mini6/images/0/0e/Lwip.pdf`, 2001.

[21] A. Dunkels. uIP – A Free Small TCP/IP Stack. `http://www.dunkels.com/adam/download/uip-doc-0.6.pdf`, 2002.

[22] A. Dunkels. Rime – a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands*. Citeseer, 2007.

[23] A. Dunkels, B. Grönvall, and T. 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.

[24] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. ACM, 2006.

[25] C. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A Modular Network Layer for Sensornets. In *Proc. of OSDI*, pages 249–262. USENIX Association, 2006.

[26] E. Engel. added incomplete RPL. `https://github.com/RIOT-OS/RIOT/commit/550c829f0d38d32a0952479950bce58d0bd3cd76`, 2012. [Online] Accessed: 2016-05-24.

[27] Ericsson. More than 50 billion connected devices. Technical report, Ericsson, 2011.

[28] ETSI. 6LoWPAN Plugtests. Technical report, 2013. https://portal.etsi.org/CTI/Downloads/TestReports/6LoWPAN_Plugtest_TestReport_v100.pdf.

[29] FIT IoT-LAB. M3 Open Node – Data Sheet. https://www.iot-lab.info/hardware/m3/, 2016. [Online] Accessed: 2016-06-19.

[30] E. Fleury, N. Mitton, T. Noel, C. Adjih, V. Loscri, A. Vegni, R. Petrolo, V. Loscri, N. Mitton, G. Aloi, et al. FIT IoT-LAB: The largest iot open experimental testbed. *ERCIM News*, 2015(101), 2015.

[31] Free Software Foundation, Inc. lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip, 2016. [Online] Accessed: 2016-06-19.

[32] Free Software Foundation, Inc. Make. https://www.gnu.org/software/make/, 2016. [Online] Accessed: 2016-05-22.

[33] Freescale Semiconductor, Inc. *Kinetis KW41Z/31Z/21Z Wireless MCUs - Fact Sheet*, 2015.

[34] Freie Universität Berlin. MSB A2. http://www.mi.fu-berlin.de/inf/groups/ag-tech/projects/Z_Finished_Projects/ScatterWeb/modules/mod_MSB-A2.html, 2008. [Online] Accessed: 2016-05-24.

[35] O. Gesch. Evaluation of TCP as Transport Layer Protocol for LoWPANs. Master Thesis, 2011.

[36] O. Hahm, K. Schleiser, H. Petersen, M. Lenders, P. Kietzmann, E. Baccelli, T. Schmidt, and M. Wählisch. RIOT: Reconsidering Operating Systems for Low-End IoT Devices. May 2016.

[37] J. Hennessy and D. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[38] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), 2006. http://tools.ietf.org/html/rfc4291.

[39] Hochschule Offenburg. *Documentation of the emb6 Network Stack*, v0.1.0 edition, 2015.

[40] J.-H. Hoepman and B. Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

[41] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), 2011. http://tools.ietf.org/html/rfc6282.

[42] N. Hutchinson and L. Peterson. Design of the x-kernel. In *Proc. of ACM SIGCOMM*, pages 65–75. ACM, 1988.

[43] ICANN. Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied. https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf, 2011.

[44] Y. Joo-Sang and H. Yong-Geun. Transmission of IPv6 Packets over Near Field Communication. Internet Draft, 2015. http://tools.ietf.org/html/draft-ietf-6lo-nfc-02.

[45] E. Kim, D. Kaspar, and J. Vasseur. Design and Application Spaces for IPv6 over Low-Power Wireless, Personal Area Networks (6LoWPANs). RFC 6568 (Informational), 2012. https://tools.ietf.org/html/rfc6568.

[46] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), 2006. http://tools.ietf.org/html/rfc4340.

[47] M. Kovatsch, S. Duquennoy, and A. Dunkels. A low-power CoAP for Contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860. IEEE, 2011.

[48] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919 (Informational), 2007. http://tools.ietf.org/html/rfc4919.

[49] LAN/MAN Standards Committee and others. Part 15.4: Low-rate wireless personal area networks – IEEE 802.15.4-2011. https://standards.ieee.org/getieee802/download/802.15.4-2011.pdf, 2011.

[50] S. Lembo, J. Kuusisto, and J. Manner. In depth breakdown of a 6LoWPAN stack for sensor networks. *International Journal of Computer Networks & Communications (IJCNC)*, 2(6), 2010.

[51] M. Lenders. Implementierung eines Border Routers für 6LoWPAN unter dem μkleos-Betriebsystem. Bachelor Thesis, 2011.

[52] M. Lenders. core: msg: update detail section on IPC. https://github.com/RIOT-OS/RIOT/commit/512448ba122ad80e748eb946ed9c86ffc2e3e3f2, 2015. [Online] Accessed: 2016-06-01.

[53] M. Lenders. Add minimum payload length. https://github.com/authmillenon/RIOT_playground/commit/9d4cbb2a7ce8f60f586fe89524c6afc4a54de049, 2016. [Online] Accessed: 2016-06-04.

[54] M. Lenders. lowpan6: wrong datagram size for fragmentation. http://savannah.nongnu.org/bugs/?47291, 2016. [Online] Accessed: 2016-06-19.

[55] M. Lenders. Merge pull request #5450 from cgundogan/pr/backport/make/enable_nhc. https://github.com/RIOT-OS/RIOT/commit/20ba06257b48ac1a990e27136b188603286e3bd9, 2016. [Online] Accessed: 2016-06-04, HEAD of 2016.04 branch at time of experimentation.

[56] P. Levis. Experiences from a Decade of TinyOS Development. In *Proc. of OSDI*, pages 207–220. USENIX Association, 2012.

[57] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[58] K. Lynn, J. Martocci, C. Neilson, and S. Donaldson. Transmission of IPv6 over MS/TP Networks. Internet Draft, 2015. `http://tools.ietf.org/html/draft-ietf-6lo-6lobac-03`.

[59] S. Madakam, R. Ramaswamy, and S. Tripathi. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 3(05):164, 2015.

[60] P. Mariager, J. Petersen, Z. Shelby, M. van de Logt, and D. Barthel. Transmission of IPv6 Packets over DECT Ultra Low Energy. Internet Draft, 2015. `http://tools.ietf.org/html/draft-ietf-6lo-dect-ule-03`.

[61] Y. Mazzer and B. Tourancheau. Comparisons of 6LoWPAN Implementations on Wireless Sensor Networks. In *Proc. of SENSORCOMM*, pages 689–692, 2009.

[62] P. Middleton, P. Kjeldsen, and J. Tully. Forecast: The Internet of Things, worldwide, 2013. *Gartner Research*, 2013.

[63] N. Möller, K. Johansson, and H. Hjalmarsson. Making retransmission delays in wireless links friendlier to TCP. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 5, pages 5134–5139. IEEE, 2004.

[64] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), 2007. `http://tools.ietf.org/html/rfc4944`.

[65] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), 2007. `http://tools.ietf.org/html/rfc4861`.

[66] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby, and C. Gomez. IPv6 over BLUETOOTH(R) Low Energy. RFC 7668 (Proposed Standard), 2015. `http://tools.ietf.org/html/rfc7668`.

[67] K. Nithin. BLIP: An implementation of 6LoWPAN in TinyOS, 2010.

[68] H. Petersen. cosy – Python tool analyzing memory usage and distribution in .elf files. `https://github.com/haukepetersen/cosy`, 2015. [Online] Accessed: 2016-06-19.

[69] H. Petersen, M. Lenders, M. Wählisch, O. Hahm, and E. Baccelli. Old Wine in New Skins?: Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices. In *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*, pages 31–35. ACM, 2015.

[70] Raspberry Pi Foundation. Raspberry Pi – Teach, Learn, and Make with Raspberry Pi. `https://www.raspberrypi.org`, 2016. [Online] Accessed: 2016-06-19.

[71] D. Ritchie. The UNIX System: A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[72] U. Sarwar, G. Rao, Z. Suryady, and R. Khoshdelniat. A comparative study on available IPv6 platforms for wireless sensor network. *World Academy of Science, Engineering and Technology*, 62:889–892, 2010.

[73] K. Schleiser. core: mbox: initial commit. `https://github.com/RIOT-OS/RIOT/pull/4919`, 2016. [Online] Accessed: 2016-06-19.

[74] A. Shah, H. Acharya, and A. Pal. Cells in the Internet of Things. *arXiv preprint arXiv:1510.07861*, 2015.

[75] Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2009.

[76] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775 (Proposed Standard), 2012. `http://tools.ietf.org/html/rfc6775`.

[77] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), 2014. `http://tools.ietf.org/html/rfc7252`.

[78] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), 2007. `http://tools.ietf.org/html/rfc4960`.

[79] T. Jaffey. microcoap. `https://github.com/1248/microcoap/`, 2013. [Online] Accessed: 2016-05-22.

[80] A. Tanenbaum and H. Bos. *Modern Operating Systems*. Prentice Hall Press, 4th edition, 2014.

[81] J. Tanenbaum, A. Williams, A. Desjardins, and K. Tanenbaum. Democratizing technology: pleasure, utility and expressiveness in DIY and maker practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2603–2612. ACM, 2013.

[82] Texas Instruments. *CC1100 – Low-Power Sub- 1 GHz RF Transceiver*, 2009.

[83] Texas Instruments. *TI SimpleLink CC3000 Module – Wi-Fi 802.11b/g Network Processor*, 2012.

[84] The FreeBSD project. FreeBSD Ports. `https://www.freebsd.org/ports/`, 2016. [Online] Accessed: 2016-05-22.

[85] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), 2007. `http://www.ietf.org/rfc/rfc4862.txt`.

[86] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister. OpenWSN: a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.

[87] R. Weber. Internet of Things – New security and privacy challenges. *Computer Law & Security Review*, 26(1):23–30, 2010.

[88] H. Will, K. Schleiser, and J. Schiller. A real-time kernel for wireless sensor networks employed in rescue scenarios. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 834–841. IEEE, 2009.

[89] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), 2012. `http://tools.ietf.org/html/rfc6550`.

[90] J. Ye. GCC ARM Embedded. `https://launchpad.net/gcc-arm-embedded`, 2011. [Online] Accessed: 2016-06-19.

[91] C. Yibo, K.-M. Hou, H. Zhou, H.-L. Shi, X. Liu, X. Diao, H. Ding, J.-J. Li, and C. de Vaulx. 6LoWPAN Stacks: A Survey. In *Proc. of WiCOM*, 2011.

[92] Z-Wave Alliance. Catalog of Certified Z-Wave Products. `http://products.z-wavealliance.org/`, 2016. [Online] Accessed: 2016-06-19.

[93] S. Zeisberg. Implementierung von 6LoWPAN in μkleos. Bachelor Thesis, 2011.

[94] D. Ziegelmeier. Initial import of Ivan Delamer's 6LoWPAN implementation with slight modifications to allow compiling in IPv6 only mode. `http://git.savannah.gnu.org/cgit/lwip.git/commit/?id=e2a3565971497a8f1dabf64ba17e9547f810d219`, 2016. [Online] Accessed: 2016-06-19.

[95] ZigBee Alliance. Certified Products — The ZigBee Alliance. `http://www.zigbee.org/zigbee-products-2/`, 2016. [Online] Accessed: 2016-06-19.