NNT : 2016SACLX090

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À ECOLE POLYTECHNIQUE

École doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat: Informatique
par

M. Oliver Hahm

ENABLING ENERGY EFFICIENT SMART OBJECT NETWORKING AT INTERNET-SCALE

Experimental Tools, Software Platform, and Information-Centric Networking Protocols

Thèse présentée et soutenue à Berlin, le 01 décembre 2016.

Composition du Jury :

M.	JOCHEN H. SCHILLER	Professeur Freie Universität Berlin	(Président du jury)
M.	CARSTEN BORMANN	Professeur Universität Bremen	(Rapporteur)
M.	DIRK KUTSCHER	Docteur Huawei German Research Center	(Rapporteur)
M.	ANIS LAOUITI	Maître de conférences Telecom SudParis	(Examinateur)
M.	EMMANUEL BACCELLI	Professeur INRIA	(Directeur de thèse)

This dissertation is the result of a long, exciting, sometimes cumbersome, often surprising journey. It is dedicated to my grandmother Erna who inspired and motivated me to begin this journey. It is also dedicated to my parents, Doris and Dieter, whose love and unconditional support made this journey possible. And finally, it is dedicated to Judith who was my solace and light in the darkest nights during this journey.

Acknowledgments

During my journey towards this dissertation, I was incredibly lucky to meet many supporting and inspiring people. I would not have been able to finish this journey without the help of all of them.

First and foremost, I would like to thank Emmanuel Baccelli who supported and helped me in many ways. His guidance, expertise, and experience, but also his kindness were invaluable for me.

I am also very grateful to my reviewers, Carsten Bormann and Dirk Kutscher who were willing to read and examine this manuscript. Their amazing work in the IETF and IRTF inspired me in various aspects during the last years and I am very glad that they accepted to review this thesis.

I would also like to thank Thomas Schmidt and Matthias Wählisch for many stimulating discussions on computer science and beyond. Their astonishingly detailed and constructive feedback as well as the many good advices throughout the last years were highly appreciated.

A huge part of this work has been done in practical work on IoT and particularly with RIOT. Hence, I am thankful for all the fantastic work that has been done on RIOT and even more important: for this amazing community. Especially, I would like to thank Kaspar Schleiser who started the RIOT and was the best office mate I could imagine. I also thank Hauke Petersen for his tremendous work on many aspects of RIOT and friendship, Martine Lenders for her passionate work on RIOT's networking capabilities, Cenk Gündoğan for his dedication and curiosity, and Thomas Eichinger for his helpfulness and support on many low-level aspects. One person who certainly did an outstanding job on RIOT which enabled many great projects on RIOT and big parts of this thesis is my dear friend Ludwig Knüpfer. I appreciate that he was not only always available for a good advice, but also that he is such a great pal! My thanks go also to Peter Kietzmann, Francisco Acosta, Joakim Nohlgård, Lotte Steenbrink, René Kijewski, Christian Mehlis, Martin Landsmann, Kevin Roussel, Johann Fischer, Andreas Pauli, Stephan Zeisberg and all the other people who helped to make RIOT such a great project! Furthermore, I would like to thank Peter Schmerzl for his seminal work on operating systems which inspired huge parts of RIOT's initial design.

I thank all the many friendly colleagues I could work with at Freie Universität Berlin and at INRIA. Especially, I want to thank my friend Felix Shzu-Juraschek for encouraging me to return to the university, for his contagious cheerfulness, and his constant willingness to help. Thousand thanks go to Michael Frey for his friendship, many, many good debates, and his ability to think outside the box. I am also thankful that I had the opportunity to work with Sebastian Trapp who took me on

another (delicious) journey. Further thanks go to Mesut Günes for his advice and guidance during my first steps in research on wireless multi-hop networks. I would also like to thank Jochen Schiller for letting me work in his group and hosting me as a guest during the last four years. Big thanks go also to Stephan Adler, Stefan Pfeiffer, Alexandre Abadie, and Cédric Adjih for many good discussions and the great team play. It was a big pleasure to work with you.

Many thanks go also to all the great people I had the pleasure to work with during the G-LAB and the SAFEST project, at the IETF/IRTF, and other cooperations. I would like to especially thank Barbara Staehle for the nice collaboration and her spontaneous help in the final phase of this dissertation; Thomas Watteyne for his help and fantastic work on OpenWSN and 6TiSCH; Simon Duquennoy and Nicolas Tsiftes for the interesting exchange on IoT operating systems; Alexander Aring and Stefan Schmidt for their help and dedication to open source IoT communication. A big thank goes also to Gaëtan Harter, Frédéric Saint-Marcel, Julien Vandaele, and the rest of the great crew behind the IoT-Lab testbed who were always ready to help.

Last, but most certainly not least, I want to thank all my friends, particularly Daniel Mösinger and Pascal May. I cannot express how grateful I am for your loyalty and your support!

And finally: this one is dedicated to all the ravers in the nation.

'Why do beautiful songs make you sad?'
'Because they aren't true.'
'Never?'
'Nothing is beautiful and true.'

Jonathan Safran Foer

Contents

Glossary

6LoWPAN Border Router The 6LoWPAN Border Router is a router in a 6LoWPAN network that connects the LoWPAN to the Internet. Thus, it has typically at least two interfaces, e.g. a IEEE 802.15.4 radio and an Ethernet interface. 31, 38, 48, 153, 201

DES-Testbed The DES-Testbed is a hybrid wireless network located on the campus of *Freie Universität Berlin* with 60 nodes scattered in a realistic manner. All nodes consist of a wireless mesh router equipped with multiple IEEE 802.11a/b/g/h transceivers and a MSB-A2 sensor node. Being a standalone testbed until 2016, it became part of the FIT IoT-LAB in July 2016. xii, 86, 88, 115, 116, 118, 121, 124, 137, 147

FIT IoT-LAB The FIT IoT-LAB is a large-scale testbed for research on Internet of Things. It consists of over 2,700 wireless nodes spread over seven different sites in France and Germany. Different hardware platforms are available. xi, xii, 11, 14, 86, 88, 100, 115, 119–121, 168, 174, 183

GitHub GitHub is a web-based Git repository hosting service in conjunction with several collaboration features. It serves also as some kind of social network for open source developers. 46, 47, 50, 86, 112

GNRC RIOT's default network stack (status as of 08/2016) with support for Ethernet, IEEE 802.15.4, IPv6, UDP, and RPL. xii, 15, 92, 96, 100, 103–108, 110, 113, 197

IEEE 802.15.4 IEEE 802.15.4 is a specification defining physical and link layer technologies. Originally defined for Personal Area Networks (PANs), it has become relevant in other use cases such industrial automation or smart metering. In the basic configuration a CSMA MAC is used. The frame size is limited to 127 byte, the data rate is maximum 250 kbit/s, and transmissions are disseminated over 26 channels in sub-GHz and the 2.4 GHz ISM band . xii, 4–6, 8, 10, 11, 26, 28–30, 39, 46, 48, 71, 101, 154, 177, 180, 182

Information-Centric Networking Information-Centric Networking (ICN) is a networking paradigm deviating from current host-centric networks by focussing on *named information* (or *named data* or *content*). vii, ix–xii, 14–16, 50, 51, 107, 108, 141–143, 145, 150, 152, 156, 157, 159, 176–180, 182–184, 186, 188–191, 194, 195, 199, 203, 210–212, 226, 228

IoT-LAB-M3 A hardware platform deployed in the FIT IoT-LAB testbed. It is equipped with a 32-bit ARM Cortex-M3 microcontroller, 64 kB of RAM, 256 kB of ROM, an IEEE 802.15.4 radio transceiver, and four different sensors (light, accelerometer, gyrocope, pressure). 21, 59, 90, 96, 97, 100, 104, 156, 168, 170, 183

MSB-A2 ScatterWeb Modular Sensor Board A2, a hardware platform developed at Freie Universität Berlin and deployed in the DES-Testbed. Equipped with a 32-bit ARM7 microcontroller, 96 kB of RAM, 512 kB of ROM, the TI CC1100 radio transceiver using a proprietary Sub-GHz technology, a SHT11 temperature and humidity sensor, and the LTC4150 coulomb counter. 132–135, 155, 156, 197

netapi GNRC's central, IPC based API to communicate between the modules. 96, 103–106, 108, 113, 191

Pending Interest Table The PIT is a data structure in ICN that stores information about generated and forwarded Interests. 50, 143–145, 150, 159, 164, 179, 181, 205

Chapter 1

Introduction

Tiny devices equipped with microcontrollers (MCUs) and transceivers directly connected to the physical world are often called *Smart Objects*. The emerging Internet of Things (IoT) aims to seamlessly integrate these usually resource-constrained, often battery-operated communication devices into the global Internet. Smart Objects are often connected over low-power and lossy links. As a consequence, the heterogeneity of both hosts and links in the Internet is largely increased.

One end of the IoT is composed by these Smart Objects which directly interact with the physical world, e.g., by controlling engines or sensing the temperature. The other end is composed by powerful servers that can act as the back-end, e.g., by providing a management web interface or a database to store sensor values. Enabling end-to-end connectivity between these devices with a direct coupling to the physical world and services or users in the Internet, creates a quantity of new application domains, but also induces a number of new challenges.

Smart Objects are the key for many so-called *Smart Services*. In factory automation, for instance, direct access to physical devices enables much shorter reaction times and collecting much more fine-grained information about the system. In this manner resources can be used more efficiently than in legacy systems where data had to be processed by a central entity or offline.

However, the IoT also poses many new challenges for software architectures and network protocols which are required to operate on and between these Smart Objects. On the one hand, these devices are neither capable of running well-known and mature operating systems (OSs) such as Linux or BSD nor using traditional Internet protocols such as IPv4 or HTTP. On the other hand, providing standardized Application Programming Interfaces (APIs) and protocols is important to ensure interoperability between different vendors as well as compatibility to existing Internet systems.

1.1 From to WSN to IoT

The first research area focussing on distributed systems composed by heavily resource-constrained devices connected over unreliable, low-power wireless links was Wireless Sensor Networks (WSNs). This research area emerged from the

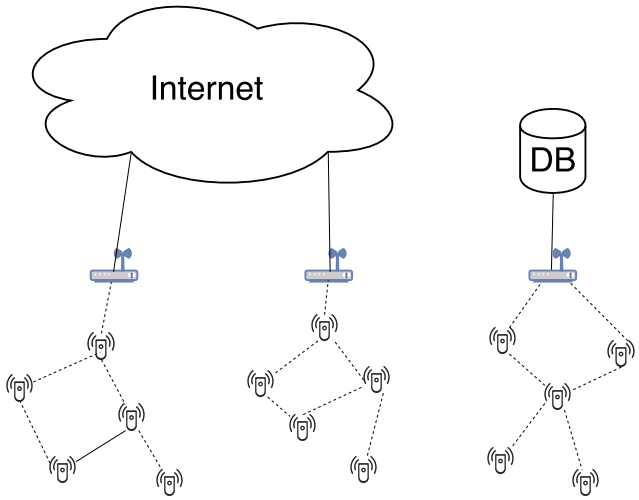


Figure 1.1: Typical WSN scenarios where a dedicated sink is either connected to the Internet or a database.

Smart Dust research project, which was started in 1998 at the University of California in Berkeley [Kahn et al., 1999]. For almost two decades, a variety of protocols and networking algorithms, addressing the peculiarities and limitations of these systems, has been designed, implemented, and evaluated. A plethora of articles analyzing theoretical models, discussing results from simulations, and testbed driven research has been published [Karl and Willig, 2007, Khan et al., 2016]. At the same time, researchers in this area developed the necessary tools and software components to examine these systems. Along with the implementation of protocols, algorithmic frameworks, simulators and emulators, or experimentation libraries, a variety of OSs and middleware for typical WSN use cases has been implemented [Moubarak and Watfa, 2009, Rawat et al., 2014].

Over time proprietary protocol stacks, such as ZigBee or WirelessHART, have emerged as default solutions in commercial scenarios [Gungor and Hancke, 2009]. Typical use cases for WSNs comprise agriculture control, disaster prevention systems, wildlife or structural health monitoring [Akyildiz and Vuran, 2010]. Convergecast is the typical communication pattern in most of these WSN scenarios where sensor values are forwarded towards a single data sink, e.g., a more powerful gateway node connected to the Internet or a database, as depicted in Figure 1.1. Consequently, each WSN deployment is typically controlled by a single entity through the gateway. This entity also has full governance over the network and data inside the WSN. Typically, these networks are deployed once and further nodes are added only to replace malfunctioning ones or to react to changes in the task dynamics [Akyildiz et al., 2002].

New Hardware and new Business Models

In the meantime, not only the algorithms and protocols reached a certain level of practicability, but also the hardware in this area has evolved to fulfill the requirements of industrial and commercial deployments:

- MCUs that are not only small and energy efficient enough, but are also available for a very low price.
- Digital radio transceivers that are comparatively simple to program and provide energy-saving RX and TX operations.
- A huge variety of cheap sensors to measure a wide range of physical properties.

However, it is not expected that new generations of hardware in this area will relax computational or memory constraints: IoT devices are rather expected to get smaller, cheaper, and more energy efficient [Mirani, 2014].

The innovations in hardware design and algorithms enabled the emergence of many new business models. Additional to the traditional WSN deployments, multiple new use cases such as smart building, smart homes, industrial automation, smart metering, and smart grid have been identified for potential business strategies. But also operating areas with very different requirements such as health care, aerospace industry, or city governments became potential beneficiaries of these technologies.

The heterogeneity of use cases, hardware, and communication protocols, the huge number of stakeholders, and the pervasive and persistent nature of the IoT engender a set of new opportunities and challenges

All these commercial, industrial, and governmental use cases also created the requirement for standardization efforts. On the one hand, it has become increasingly important that devices from different vendors are interoperable on various layers of the network stack. On the other hand, a similar need for compatibility has arisen for software components. Gradually, it has become clear that there is a need for natural, seamless interconnection to the worldwide network infrastructure: the vision of the Internet of Things emerged [Ko et al., 2011]. Consequently, different standardization bodies such as the Institute of Electrical and Electronics Engineers (IEEE), Internet Engineering Task Force (IETF), Open Mobile Alliance (OMA), or Object Management Group (OMG) have tackled this task and released corresponding protocol specifications. The IETF, for instance, has introduced a set of standards: IPv6 over IEEE 802.15.4 networks (6LoWPAN) [Montenegro et al., 2007, Hui and Thubert, 2011] (an adaptation layer which

compacts long IPv6 headers so they fit in short frames typical for sensor networks—like IEEE 802.15.4 frames), Routing Protocol for Low-Power and Lossy Networks (RPL) [Winter et al., 2012] (a routing protocol) and Constrained Application Protocol (CoAP) [Shelby et al., 2014] (an application-layer protocol allowing low-power devices to appear as web servers).

These emerging standards also function as a catalyst for more commercial use cases. Companies which refrained from using these new technologies got attracted by the availability of standard solutions. Networking systems with Internet Protocol (IP) at the narrow waist of the protocol stack has proven to work fine for a large variety of use cases over the last 40 years. Consequently, using IP suite protocols to allow for end-to-end connectivity between low-end devices on the one side and traditional Internet services at the other side enabled a whole set of new business cases.

In contrast to WSNs scenarios, IoT deployments comprise a much larger heterogeneity. Not only in terms of hardware and link layer technologies, but also in terms of network configurations and applications. In comparison to WSNs where separated networks, each tailored for one particular use case, were deployed, IoT applications are composed of several components that are supposed to seamlessly work together—either locally in an ad-hoc manner between these IoT devices, interconnected through full-fledged wired backbone technologies, or even through cloud services. IoT deployments are furthermore supposed to be deployed for a long and often undetermined timespan. Where the size of WSN deployments is usually not subject to bigger changes [Akyildiz et al., 2002], IoT deployments are expected to be gradually extended and updated over time. Therefore, a need for standards, such the Internet protocol suite, is inevitable.

1.2 Use Cases

The IoT comprises a wide range of professional use cases [Gubbi et al., 2013]. In contrast to WSN scenarios these use cases comprise many different communication and traffic patterns [Tschofenig et al., 2015]. Therefore, it is important to take a closer look at some use cases for IoT applications in order to understand the variety of scenarios and derive the particular requirements. This section thus presents four very different use cases. For each use case this section presents

- typical applications,
- an example scenario,
- the limitations of legacy systems that could be overcome by deploying IoT solutions, and
- the key challenges for an IoT deployment in this area.

Industrial Automation

Applications Steel mills, oil refineries, chemical industries, or power plants are examples for industrial settings where complex monitoring and management processes occur. Sensor values such as temperature, pressure, vibration, or tank fill levels are used to control actuators and coordinate production stages often by thousands of nodes.

The particular requirements of these applications for industrial networks in contrast to the traditional Internet, have lead to completely decoupled development of technologies, protocols, and standards. While the Internet is built to interconnect billions of heterogeneous devices communicating globally large amounts of data, an industrial network is typically deployed within a factory floor, typically connecting 100's or 1000's of devices. In many cases the amount of traffic and content data in industrial applications is not very large, but reliability, dependability, and deterministic latencies are often mandatory.

An Example Scenario The example scenario[1] is a factory automation application using a combined IEEE 802.15.4 and Power Line Communication (PLC) network. Networks of large scale with up to 8000 nodes, a high density, and many line-of-sight connections (maximum three hops) are expected. Bi-directional transmissions with $\approx$ 10 byte payload and a datarate $>$ 100 kbit/s are required. An end-to-end latency below 100 ms and clock synchronization with an error below 100 µs are further requirements. All nodes are battery powered and should be reprogrammable over the air. Secure communication and automatic commissioning are required, hardware encryption support is available. Nodes have 128 kB ROM and 20 kB RAM.

Limitations of Legacy Systems Traditionally, wired networks have been used to address the requirements of industrial systems [Decotognie and Pleinvevaux, 1993]. However, wired solutions are not always feasible. In many cases, the cost of wiring caused by planning and installation is the prohibitive factor. Explosive environments and hot surfaces, e.g. in a refinery, are other restrictions for wired solutions. Finally, mobile devices cannot be connected in a wired manner at all.

As a consequence, wireless technologies have become increasingly appealing for industrial application, reducing installation costs tremendously. In order to provide wire-like reliability, and offer operation without maintenance needs over several years eventually lead to specialized wireless technologies such as

[1] This example is taken from a company using RIOT for their business model.

WirelessHART [HART Communication Foundation, 2008]. However, these specialized solutions prevent seamless interoperability with other deployed systems which often have different requirements.

Key Challenges The manifold requirements of industrial applications demand for the deployment of many diverse technologies. These different deployments have to be seamlessly interoperable. Moreover, extensions and upgrades must be possible at any point of time. Particular for low-power, wireless networks, industrial applications feature some particular requirement and challenges. Networks often have to operate under harsh conditions where all kind of machines can cause bursty interferences or the node density is enormously high. Often these deployments have also very strict requirements in terms of reliability—i.e. almost no packet loss can be tolerated—, and latency—i.e. packets have strict deadlines. Finally, the commercial nature of these deployments and safety considerations require a high degree of security [Pister et al., 2009].

Mobile Health

Applications In mobile health applications the goal is typically to monitor a patient with various devices [Pantelopoulos and Bourbakis, 2010]. These devices are equipped with one or more sensors to measure, for example, the blood pressure or the heart rate. Typical applications are hearing aids, drug dosage, ECG[2], EEG[3], temperature, respiration, or glucose monitoring. Mobile health applications can help to reduce the exploding costs for health care, caused by an aging population and sedentary life style. At the same time they can help to improve the quality of living for sick or handicapped people.

An Example Scenario The example scenario is a wearable heart rate monitor, using IEEE 802.15.4j or IEEE 802.15.6 (Wireless Body Area Network (WBAN)) [Patel and Wang, 2010, IEEE802.15.6, 2008]. Networks are small with less than 12 nodes and require a radio range below 5 m. A star or tree topology is assumed. One-directional transmissions with ≈ 10 bit payload and a datarate of ≈ 100 bit/s are required. An end-to-end latency of less than 250 ms is required and the network should be able to setup a new link in less than 3 s. The bit error rate must be below 10^{-10}. All nodes are battery powered and energy harvesting techniques are desirable. Security and privacy are a top priority. Nodes have 48 kB ROM and 10 kB RAM.

[2] Electrocardiogram
[3] Electroencephalogram

Limitations of Legacy Systems There are two main problems with traditional health monitoring devices, such as Holter monitors [Milenković et al., 2006]:

1. Since the data is processed offline, they are not able to perform continual online monitoring and early detection of medical disorders.
2. They limit the patient's activity and level of comfort, which influences the measured results.

These systems have either no communication subsystem at all or use proprietary technologies which require a dedicated gateway instead of leveraging the widespread availability of devices such as smartphones. Missing interoperability also limits the immediate cooperation between various systems.

Key Challenges In order to reduce weight and energy consumption of mobile health devices, they are required to leverage co-existing deployments, e.g., other WBANs or smartphones, as much as possible. Consequently, these systems are required to be interoperable at the lower layers as well as to enable direct Internet connectivity. Besides this need for interoperability, the main requirements for typical mobile health applications are [Pantelopoulos and Bourbakis, 2010]: **(i)** accuracy, **(ii)** reliability, **(iii)** timeliness, and **(iv)** wearability. These requirements need to be tackled two levels: on the system and on the network level.

Nano and Micro Satellites

Applications Where traditionally huge, monolithic satellites have been used, more and more decentralized satellite architectures such as fractionated spacecrafts or satellite swarms are deployed. The focus here is on satellite systems with under 100 kg mass, which are classified according to their size and mass [Rodrigo Muñoz, 2016]. Satellites with a mass between 10 and 100 kg are considered as micro-satellites. They are considered as nano-satellites if their weight is between 1 and 10 kg. In some scenarios even pico- (100 g to 1 kg) and femto-satellites (< 100 g) are relevant. Wireless systems are deployed on these small satellite systems **(i)** to replace cables inside the satellites to save weight and **(ii)** for communication among the satellites.

A typical mission for these clusters of smaller modules is earth observation. Potential future earth observation applications such as real-time videos, stereoscopic radar, or multi-spectral imagery, are probably only possible due to the cost- and time-efficient implementations of these new approaches. The reduced costs of these systems lead to an increased interest in standardizing these distributed missions.

An Example Scenario The example scenario is an intra-spacecraft network, using IEEE 802.15.4 [Sun et al., 2010, Beekema, 2011]. Networks are small to medium-sized with between 10 and 100 nodes and require a radio range up to 10 m. Intermittent one-directional transmissions for housekeeping and ADCS[4] with a few bytes of payload and a datarate of below 100 kbit/s occur. There is no hard upper bound on the end-to-end latency and failures can be tolerated as long as they are not systematic [Beekema, 2011]. Nodes have 256 kB ROM and 16 kB RAM.

Limitations of Legacy Systems Traditional monolithic satellites are big in space and weight. Consequently, the costs for launches of these satellites is much higher compared to miniaturized versions that can be often piggybacked with other missions. The distributed approach also yields better spatial resolution and shorter revisit times than a large, monolithic satellite system. Another advantage of this decentralized cluster approach is increased redundancy. The system may continue to operate even if some of the satellites fail, resulting in increased robustness and reliability.

Key Challenges A wireless system deployed in those satellite systems requires to provide a high degree of autonomy. In many cases manual administration is impossible due to the high latency between control center and the satellite. This is, for instance, the case for autonomous formation flying where the typical maneuver cycle for maintenance of the formation may be too short for ground-control [Sun et al., 2010]. The wireless network must also be robust and able to operate under harsh environmental condition. Cosmic background radiation and a spatial high density of electric on-board devices lead to a very challenging environment for a resource constrained wireless network. Finally, the system needs to work in a very energy efficient manner. These satellite systems are supposed to be deployed for several years and only a small amount of their energy supply is available for the communication subsystem [Amini et al., 2007].

Building Automation

Applications Nowadays many buildings are equipped with a variety of controllers connected to electronic or pneumatic elements. These building management systems consist of a network of sensors, actuators, controllers, and user interface devices that interoperate to provide a safe and comfortable environment while constraining energy costs. Typical applications in this area are Heating, Ventilation, and Air Conditioning (HVAC), room lighting, window shades, solar loads,

[4] Altitude Determination and Control Subsystem

physical security, fire detection, and elevator or lift systems. They can often be found in in government facilities, pharmaceutical manufacturing facilities, hospitals, or office buildings ranging in size from 10 000 to more than 100 000 m^2. The variety of sensors range from common temperature, lighting, and humidity sensors to specialized air flow and pressure or CO_2 sensors.

An Example Scenario The example scenario is a HVAC application [Martocci et al., 2010] using a proprietary sub-GHz ISM band protocol. Networks of large scale with up to 1000 routers plus up to 1000 hosts, low to medium density, and up to 10 hops (longer distances can be routed through a wired backbone) are expected. Bi-directional transmissions with ≈ 200 byte payload and datarates of at least 20 kbit/s are required. End-to-end latency for prioritized traffic should be below 120 ms. Secure communication and automatic commissioning are required. Nodes have 256 kB ROM and 16 kB RAM.

Limitations of Legacy Systems The expenses for the installation of wired building management systems are often a significant cost factor. First, planning and deployment of cable tunnels and dispensers result in increased personnel expenses. Second, the plain costs for copper wires are not negligible, either. Moreover, (additional) wireless systems can increase the flexibility of the building management system—and thereby further improve the energy efficiency of the building. Thus, prospective systems will consist of a combination of wireless and wired networks[5]. Building automation systems are often incrementally extended and updated, hence, using proprietary technologies instead of interoperable standards such as IP-based protocols increases the complexity for the integration of new systems drastically.

Key Challenges The building control systems are typically installed during the construction phase by electricians which have very little computer knowledge. Thus, these systems require to bootstrap with no or very little configuration effort during commissioning. Moreover, these systems are often deployed while the wired, Internet-connected backbone is not yet in place. Hence, autonomous, ad-hoc communication is required. Scalability is another key challenge, with networks than span over more than 1000 nodes. In contrast to many traditional WSN use cases, supporting only multipoint-to-point traffic (*convergecast*) is not sufficient. The required close interactions of many different sensors and actuators requires support of point-to-multipoint and point-to-point communication as well. Finally,

[5]The sole use of wireless systems is impracticable, since some of the actuators require a strong power supply anyway, and some systems, e.g., video cameras, require higher data rates.

Use Case	Industrial Automation	Mobile Health	Nano and Micro Satellites	Building Automation
link layer	IEEE 802.15.4e	IEEE 802.15.4j	IEEE 802.15.4	proprietary
# nodes	up to 8000	< 12	10–100	up to 2000
density	high	medium	low	low to medium
diameter	3	1	< 3	10
traffic	high	low	low to medium	high
real-time	strict	strict	relaxed	relaxed
time sync	✓	-	-	✓
OTA updates	✓	-	✓	✓
ROM	128 kB	48 kB	256 kB	256 kB
RAM	20 kB	10 kB	16 kB	16 kB

Table 1.1: Overview over the properties of the presented example applications for the four IoT use cases.

particular physical security and fire detection systems require support of prioritized traffic with bounded latencies.

Conclusions

Reviewing the four presented use cases concerning their key challenges, some common requirements can be derived. A first observation is the strong demand for **interoperability** between IoT systems. The expected permanence of most IoT deployments pose a need for standards that can be expected to remain a good basis for this interoperability for a long time. Next, it can be observed that all use cases require a high degree of **energy efficiency**. Furthermore, in all uses, the networks are required to offer a decent degree of **reliability**, i.e. to guarantee that a certain piece of information is delivered to the intended recipient. In some cases **latency** is an issue and the delivery is required to happen before a certain deadline. Missing this deadline will decrease the performance of the network or even lead to a complete failure. Moreover, the **robustness** is a challenge for a all these networks. The network must be able operate and guarantee a certain degree of availability even under harsh conditions, with co-existing other networks, and interferences. The next common requirement is **autonomy**, i.e. the network needs to be able to bootstrap and stay operable with a minimum of administration. **Security** is another fundamental requirement for all of these use cases. The final observation is that the MCUs in these use cases will rarely offer more than 500 kB of ROM or more than 100 kB of RAM. Consequently, **memory efficiency** is another goal.

The properties and numbers for the exemplary applications, as presented in Table 1.1 are taken from relevant surveys in the respective area where not stated otherwise. On the one hand, it can be observed that some key characteristics such as a low-power, wireless link layer technology or a memory-constrained device can be found in each example. On the other hand, other properties such as the size of the network, the amount of expected traffic, or the need for time synchronization vary widely. Hence, a further goal for the tools and mechanisms presented in this thesis is to be flexible and adaptive enough to cover these different scenarios.

Synopsis: Based on the analysis of use cases, this thesis focusses on enabling the large-scale deployment of energy efficient IoT applications, subject to low radio throughput, memory constraints, and reliability.

1.3 Enabling Smart Object Networking at Internet-Scale

From the hardware perspective, platforms to fulfill the IoT requirements—particularly in terms of energy efficiency—are already available as prototypes and development kits. MCUs with integrated radio transceivers consuming less than 1 µW while sleeping can be purchased for about $3 and it can be expected that their price will further drop when produced in larger quantities. IEEE and other standardization bodies have specified several link layer technologies, particularly designed for special use cases such as IEEE 802.15.6 (WBANs) or IEEE 802.15.4g (for Advanced Metering Infrastructures (AMIs)). Many of these specifications are already supported by the corresponding radio transceivers. Moreover, research on WSNs and Mobile Ad-hoc NETwork (MANET) has brought forward many networking protocols such as routing or data aggregation protocols, particularly tailored for the needs of low-power wireless networks.

However, to fully implement the IoT some building blocks are still missing.

Experimental Tools The particular communication patterns and large heterogeneity of IoT use cases require new tools and approaches for large-scale evaluation. While appropriate testbeds, such as the FIT IoT-LAB [Adjih et al., 2015], have been developed and deployed, additional tools are necessary to gain a deeper understanding of these systems. Motivations for that are **(i)** the complexity of distributed embedded systems, **(ii)** the effects on low-power wireless networks by a not fully controllable environment, and **(iii)** the limited debugging facilities for many embedded devices. Consequently, advanced emulation tools are required that allow to run the unmodified code for IoT devices in a controllable environment.

Another challenge for experiment-driven research on IoT is the analysis of energy efficiency, which has been shown to be a crucial property for IoT use cases. Conventional approaches to evaluate the energy consumption are not fine-grained enough and thus give only limited insights about the efficiency of the implementation.

Software Platform The availability of Linux as an open-source OS has been a catalyst for the success of open standards and therefore the Internet itself [West and Dedrick, 2001]. However, neither Linux nor one of its derivates, such as Android, are applicable on low-end IoT devices, because they cannot run on the limited resources provided on such hardware. Hence, it can be argued that a similar de-facto standard OS is required to fulfill the potential of IoT, providing a consistent API and SDK[6] across heterogeneous IoT hardware platforms. Such an OS would need to cater for multiple network stacks and for continuous network stack evolution. It must be flexible enough to address the need of professional IoT use cases, on the one hand, and allow to easily exchange and modify components on each layer, on the other hand.

Energy Efficient Networking Protocols Standardization bodies such as the IETF developed protocols particularly tailored for IoT use cases during the last decade. These protocols are often stripped-down versions of traditional Internet protocols, facilitating seamless integration into the Internet, but use, e.g., binary encodings instead of plaintext or XML representations However, since energy efficiency is one of the most crucial requirements for IoT scenarios—as seen in Section 1.2—it is mandatory to design protocols that do not only reduce the amount of traffic, but also allow for long sleep cycles of the devices. Such an approach must also consider cases where a gateway or proxy node is not available for most of the time. Instead of relying on dedicated nodes which are potentially more powerful, a distributed, cooperative approach is desirable. Obviously, the required control traffic for coordination must kept to a minimum here, too.

1.4 Contributions

In this thesis, I closely analyze the requirements for software and networking solutions as well as the necessary tools and methodologies to enable the deployment of energy efficient IoT solutions. Furthermore, I present the design and implementation of a general purpose OS, RIOT, that meets these requirements, and present

[6] Software development kit

an information-centric approach to address occurring trade-offs with respect to energy efficiency in the presented IoT use cases. All contributions in this thesis have been validated and evaluated by extensive experiments in large-scale testbeds and sophisticated virtualization environments. The research presented in this thesis followed an holistic approach, i.e. the proposed mechanisms were not studied isolated, but embedded in a full software and network stack.

The key contributions of this thesis are summarized as follows:

- A requirement analysis for the design of an OS to meet the particular requirements of the emerging IoT use cases. I present key design choices for such OSs from both, a technical and non-technical point of view. I survey applicable OSs, focusing on candidates that could become an equivalent of Linux for such devices i.e. a one-size-fits-most, open source OS for low-end IoT devices.
  This contribution has also been published in [12].

- *RIOT* is presented as a micro-kernel based OS that addresses the derived requirements for IoT use cases. In terms of resource frugality, RIOT is shown to be on par with Contiki and TinyOS, the two operating systems which pioneered IoT software platforms. Furthermore, RIOT offers an advantageous trade-off between high-level OS functionality and performance with low resource requirements, allowing IoT code efficiency and portability to a wide range of hardware and use cases. Lessons learnt with RIOT show that exotic programming aspects imposed by Contiki and TinyOS are not necessary on IoT devices. I present a detailed study of the RIOT micro-kernel and hardware abstraction, and the networking subsystem. I also overview non-technical aspects concerning the grassroots open source community of RIOT.
  This contribution has also been published in [8, 9, 17, 21].

- In order to gain deeper insights in the energy consumption of IoT implementations, I developed *DES-eProf*, a tool to conduct online, in-situ profiling of energy consumption on IoT systems. DES-eProf is highly integrated into the system and allows the developer of IoT algorithms to analyze the power consumption almost without any additional effort. It can break the power consumption down to any desired level from function-based or thread-based analysis to the evaluation of functional blocks. Compared to similar approaches in this field, DES-eProf does not require any modifications to device drivers or application code. Due to this easily deployable method it is feasible to examine the demand for energy for every single node in the network.
  This contribution has also been published in [3] and [6].

- In order to overcome observed trade-offs between energy efficiency and other IoT requirements I studied the feasibility, advantages, and challenges of an approach based on Information-Centric Networking (ICN) in the IoT. I report on the first ICN experiments in a life-size IoT deployment, spread over tens of rooms on several floors of a building. Several interoperable Named-Data Networking (NDN) enhancements, such as Reactive Optimistic Name-based Routing (RONR), a name-based routing protocol for IoT scenarios, are proposed and evaluated. Using standard ICN mechanisms and RONR significantly decreases control traffic (i.e. Interest messages) and leverages data path and caching to match IoT requirements in terms of energy and bandwidth constraints. This evaluation also provides the first experimental comparison of ICN with the common IoT standards 6LoWPAN, RPL, and UDP.
  This contribution has also been published in [18].

- Next, I analyzed how ICN caching strategies can be leveraged to improve content availability in IoT scenarios with sleepy nodes. Devices must sleep as much as possible to reduce power consumption in a variety of IoT scenarios. At the same time, data produced by IoT devices must remain at a high level of availability. Absent a centralized cache that never sleeps a trade-off appears between power consumption and IoT data availability. I propose a coordinated sleeping approach, called *Deputy on Watch*, along with the name-based caching and cache replacement strategy called Max Diversity Most Recent (MDMR). The evaluation contains a model- and experiment driven analysis of the trade-off between energy efficiency and content availability. Experiment results show that devices can reduce energy consumption by an order of magnitude while maintaining IoT content availability above 90 %.
  This contribution will be published in [14].

- Finally, I studied an approach to cater for scenarios with strong requirements in terms of reliability and determinism while maintaining a low energy consumption using ICN over Time-Synchronized Channel Hopping (TSCH). I show how ICN communications patterns can be leveraged to dynamically optimize the schedule computation for TSCH, a wireless link layer technology increasingly popular in the IoT. I propose an information-centric reservation mechanism. Through a series of experiments on FIT IoT-LAB, the approach is shown to be fully robust against wireless interference, and almost halves the energy consumed for transmission when compared to CSMA. Most importantly, the proposed adaptive scheduling prevents the time-slotted MAC layer from sacrificing throughput and delay.
  This contribution has also been published in [13] and [15].

1.5 Structure and Overview

The first part of this thesis reflects state of the art of IoT communication protocols and software platforms.

Chapter 2 defines the properties of the low-power wireless networks which are in scope for this thesis. Crucial requirements for protocols to function in such an environment are derived. Then, the standard protocol stack based on IPv6 and 6LoWPAN and necessary auxiliary mechanisms are presented. An outlook on other communication paradigms proposed for IoT follows. The chapter concludes with an observation of two typical trade-offs in IoT scenarios with respect to energy efficiency.

Chapter 3 defines the properties of low-end IoT devices which are in scope for this thesis. It derives crucial requirements and resulting design choices for system software and middleware to operate on these devices in IoT scenarios. Then, a wide selection of potential candidates to become the default OS for IoT use cases is presented.

The second part of this thesis is dedicated to the tools and software components that are necessary to conduct research on IoT.

The following Chapter 4 presents RIOT as an OS for reliable IoT. It gives a general overview about RIOT and its architecture, followed by a closer look on some implementation details and an analysis of its memory consumption. Then, its default network stack, GNRC, is presented in detail. Finally, I discuss the importance of the surrounding open-source ecosystem.

Chapter 5 presents challenges and limitations of testbed-driven research in IoT scenarios as well as a set of tools to schedule, execute, and evaluate experiments in IoT testbeds. Next, it discusses how virtualization of IoT hardware and networks can be used to overcome some limitations and present the corresponding tools. The last part of this chapter presents an energy-profiling approach to evaluate the energy consumption of an IoT system in a fine-grained manner.

The last part of this thesis discusses networking protocols for energy efficient and reliable IoT. It particular focusses on Information-Centric approaches.

Chapter 6 proposes ICN as a potential solution to overcome some shortcomings of the standard IP-based IoT stack. A general analysis of how ICN fits the requirements of IoT scenarios is followed by a study about the challenges for ICN in IoT. RONR is presented as a memory-efficient routing scheme for ICN IoT scenarios. The chapter concludes with a qualitative and quantitative comparison of the ICN stack to the standard 6LoWPAN stack.

Chapter 7 evaluates how ICN can support sleepy nodes in IoT scenarios. It proposes a name-based caching strategy called MDMR to improve content availability in ICN networks with sleepy nodes. I have studied scenarios with uncoordinated and coordinated sleeping cycles. Content availability in scenarios with uncoordinated sleeping is then examined in a theoretical model, followed by testbed-based experimental validation. The chapter concludes with the presentation of some auxiliary mechanisms that can help to further improve the results.

Chapter 8 discusses the benefits of an ICN deployment over the TSCH Medium Access Control (MAC) layer. It describes the potentials for a corresponding link-layer adaptation for ICN. Next, I propose to leverage ICN's symmetric traffic pattern to construct and maintain a link-layer transmission schedule. The chapter concludes with an evaluation of Packet Delivery Ratio (PDR) and energy consumption for this approach.

This manuscript ends with a conclusion and outlook in Chapter 9.

Part I

State of the Art on IoT Communication and Software

Chapter 2

IoT Communication

Communication in IoT systems is not always wireless, but almost always constrained in at least one aspect. The rest of the thesis will focus on wireless networks, but many issues are also applicable for other types of IoT communication.

This chapter will first define the term Low-power and Lossy Network (LLN) and then review and discuss the requirements of IoT communication with a special focus on energy efficiency and reliability. The following section analyzes the state of the art which is an IPv6 based network stack. It considers protocols that were proposed with a special focus on IoT networks and study their impact on energy consumption and reliability. The final section provides an outlook on alternative approaches basing on different paradigms.

2.1 Low-Power and Lossy Networks

The term LLN was proposed in the context of the IETF Routing over Low power and Lossy networks (ROLL) Working Group (WG) and formally defined in RFC7102 [Vasseur, 2014] as follows:

> Typically composed of many embedded devices with limited power, memory, and processing resources interconnected by a variety of links, such as IEEE 802.15.4 or low-power Wi-Fi. There is a wide scope of application areas for LLNs, including industrial monitoring, building automation (HVAC, lighting, access control, fire), connected home, health care, environmental monitoring, urban sensor networks, energy management, assets tracking, and refrigeration.

In the IETF context the terms "Low-power Wireless Area Networks", "Networks of Resource Constrained Nodes", or "Constrained Node Networks"[1] are used in a similar sense.

[1] More precisely, LLNs could be described as the combination of a Constrained Node Network and a Constrained Network.

The common properties of these networks are:

- **low bandwidth**
  Highly energy efficient transceivers, slow MCUs, and the need to share the medium among thousands of nodes lead to very low link layer data rates in the range of 200 kbit/s or even significantly below.
- **small packet sizes**
  Constrained memory, energy constraints, and the need not to occupy the medium for a too long period require small frame sizes.
- **significant packet loss**
  Due to the low-power transmissions and general challenges of wireless communication (such as interference and multi-path fading) a double-digit percentage for packet loss rate is not unlikely.
- **omnidirectional transmissions**
  Most of the time IoT radio transceivers transmit in an omnidirectional manner. Hence, on the physical layer all transmissions are broadcast. Multicast (if available) and unicast can only be filtered after reception of the destination address field. This can easily lead to congestion on the link layer.

It can be observed that these properties are partly induced by the wireless medium, partly by the need for maximum energy efficiency, and partly by the constraints of the nodes. Despite all these limitations and constraints many of the IoT use cases, as described in Section 1.2 aim for reliability and resource availability.

2.2 Requirements for IoT Network Protocols and Algorithms

Derived from IoT use cases, as the ones presented in Chapter 1, several key requirements for LLNs can be identified. Most of these requirements cannot be tackled by a single protocol or on a single layer of the stack, but require close cooperation between the protocols in the stack.

Interoperability
Interoperability on various layers of the network stack enables cooperative behavior between otherwise independent IoT deployments. In order to reduce redundancies and increase the efficiency, IoT deployments are required to interoperate with each other as much as possible. The vast heterogeneity of IoT systems makes it unpredictable which requirements for cooperation and interaction between various deployments may arise in the future. Hence, it is crucial to deploy a common standard basis that allows for a versatile interoperability. It can be expected that

IoT applications are deployed for very long amount of time. Consequently, deploying systems which are based on mature and generic protocols are the best way to make these systems future-proof.

Synopsis: Interoperability between nodes inside a LLN, but also between the LLNs themselves is mandatory for current IoT applications as well as for the long-term deployment goals.

Energy Efficiency

A golden rule for IoT networks: it is mandatory that a node sleeps often and transmits seldom. Even short sleep cycles have proven to have significant positive impact on the network lifetime [Feeney et al., 2014].

Traditionally in WSNs, computation has been energy-wise much cheaper than transmitting. However, with newer hardware platforms, e.g. based on ARM Cortex-M MCUs, the ratio between energy costs for computation and radio activity have changed. Modern MCUs consume a rather high amount of energy when active (i.e. when computing), while energy costs for radio activity (RX and TX) on modern radio transceivers have been decreasing over last years[2].

Hence, it becomes increasingly important to reduce packet processing complexity. At the same time, it is still mandatory to keep packet headers and payloads small, which makes compression technologies attractive. Consequently, a good trade-off between packet sizes and processing overhead for compression techniques needs to be found.

Synopsis: A node running out of energy is either permanently (if battery powered) or temporarily (for energy harvesting approaches) unavailable. Both consequences potentially harm the availability of its own content and the availability of the whole network.

Reduced Control and Data Traffic

In order to avoid congestion, the overall number of network-wide transmissions needs to be kept low. Hence, it is mandatory to deploy protocols that keep control traffic to a minimum. Piggy-backing additional control information on data transmissions is one way to reduce traffic. In some cases, sending fewer, but bigger packets is preferable over sending more packets [Feeney and Nilsson, 2001].

Another way to reduce packet transmissions in wireless IoT scenarios can be achieved by leveraging the broadcast nature of the medium. Nodes can overhear

[2]The MCU of the IoT-LAB-M3, for example, consumes 70 mA, when running at full speed. The MCU of the TelosB consumes only 4 mA at full speed. The radio of the IoT-LAB-M3 consumes 11.8 mA or 13.8 mA for RX respectively TX. The radio of the TelosB consumes 18.8 mA and 17.4 mA for the same modes.

traffic from their neighbors, even if none of its interfaces is addressed. Overhearing can significantly reduce transmissions and hence, energy consumption of a Wireless Mesh Network (WMN) [Iima et al., 2009].

Synopsis: A congested network or overloaded nodes harm the availability of data and services.

Bounded Latency

Several IoT use cases, as depicted in Section 1.2, require fixed upper bounds for transmission latencies. In some cases, these bounds are given by strict deadlines and require hard real-time properties from the network, while other scenarios have more relaxed requirements where delayed packets only decrease the performance. In some cases additional guarantees of minimum latency and tight jitter are also required [10]. The IETF chartered a WG called Deterministic Networking (detnet) to analyze and address this topic [Finn and Thubert, 2016].

These requirements on timings can either occur on a per-hop basis or need to be fulfilled end-to-end. While hop-wise latencies can be achieved by reservation mechanisms on the MAC layer, achieving bounded end-to-end delays requires cooperation between link layer, transport layer, and routing protocols. As an additional requirement, the network stack needs to provide an interface to the application so it can signal and specify its latency requirements.

If a limited end-to-end latency is required, packets have a certain lifetime. This lifetime must be either carried inside the packet's metadata (e.g. a header) or known by a receiving node through an additional mechanism. In some scenarios it might be advisable or even mandatory to drop packets that have exceeded their lifetime.

In order to fulfill these end-to-end latency guarantees, it is also important to introduce mechanisms that allow prioritization of certain packets or traffic flows.

Synopsis: Data may be rendered useless or applications may fail if a certain amount of packets do not arrive in time.

Robustness

Wireless communication is often harmed by significant packet loss. Packet loss due to interference may be caused by conflicting links inside the network (*intra-path* or *inter-path* interference) as well as by external effects. Certain link and MAC layer mechanisms may help to reduce or even completely eliminate internal interferences and make the network more robust against external effects [Doherty et al., 2007]. But even though these mechanisms can help to reduce packet loss, protocols and algorithms on all layers must be able to deal with packet loss.

In addition to unstable, unreliable links, the topological neighbor of a node may also change—in some cases even frequently. Certain peers may disappear or reappear, new hosts can join the networks, others may die. In mobile networks the neighborhood may be subject to continuous changes and similarly networks with sleeping nodes may encounter drastic changes to their neighborhood.

Synopsis: Protocols must be robust against drastic changes of the link quality and high dynamics of the topology.

Security and Privacy

Security is mandatory for all IoT scenarios. Achieving the classical goals of security—**(i)** confidentiality, **(ii)** integrity, **(iii)** availability, and **(iv)** non-repudiation—always requires cryptography. However, achieving a good level of secure cryptography is a challenging issue itself, but gets even more challenging in IoT scenarios due to the limited hardware capabilities and limited packet sizes. Complex computations are not feasible within reasonable time [Sethi et al., 2016] and encryption and signing adds significant overhead to the transmitted data.

Even more attack vectors exist for use cases with wireless communication. An attacker can overhear the communication between devices without any physical access to the devices or infrastructure. This does not only facilitate the overhearing possibilities for the attacker, but makes it also much harder for the network to detect the attack.

As presented in Section 1.2 the machine-to-machine communication is often part of a mission-critical element for the application. Consequently, a successful attack on the IoT communication can create severe dangers and may result in financial loss or even endanger human life. Furthermore, even in less critical scenarios, the pervasive nature of IoT makes privacy a major concern.

While new algorithms and cryptographic support on the hardware helps to enable encryption and decryption even on constrained IoT hardware, the question of a secure, but automatic key distribution still remains a problem. Pre-Shared Keys (PSKs) are often not a viable solution and can cause other vulnerabilities—especially if the attacker can physically access the hardware and memory is not protected. A prominent attack against this has been published in the context of one of the first home automation IoT products, the LIFX light bulb [Chapman, 2014]. The potential dangers of Internet connected power plugs were presented in [Gavrilut et al., 2016].

Synopsis: An attacker may harm the availability of resources, may tamper with the provided data, or significantly decrease the performance of the network.

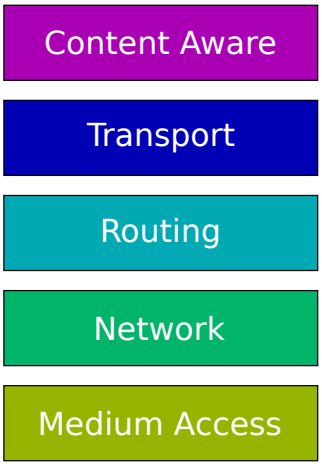


Figure 2.1: A simplified layer architecture for an LLN stack.

Summary

Many of the requirements for LLNs can be found for typical WSN applications, too. It is, however, the vast heterogeneity and strong interweaving of IoT systems that poses a new challenge here. Systems equipped with considerably different capabilities, from the hardware and from the networking perspective, running a variety of different applications under the governance of various stakeholders need to seamlessly interoperate. In contrast to WSN, IoT scenarios comprise a larger diversity of traffic patterns and are expected to be reconfigurable and extendable after deployment. IoT devices could be poured into concrete and expected to be still operative after ten years, but may be required to run a completely different application then.

2.3 Core Mechanisms for LLNs

The particular challenges of LLNs and IoT scenarios must be tackled at various layers of the network stack. Figure 2.1 depicts a simplified layer architecture derived from the traditional DoD internet architecture model [Cerf and Cain, 1983]. It identifies the following layers: **(i)** Content-aware, **(ii)** Transport, **(iii)** Network, and **(iv)** Medium Access mechanisms. Each layer can provide certain mechanisms to address the requirements of LLNs.

Content-aware mechanisms leverage their knowledge about the carried payload and request appropriate services from the lower layers. Transport mechanisms offer end-to-end connectivity and various levels of reliability to the upper layers. Network mechanisms provide energy efficient bootstrapping, Neighbor Discovery (ND) services, as well as route discovery and maintenance. Medium

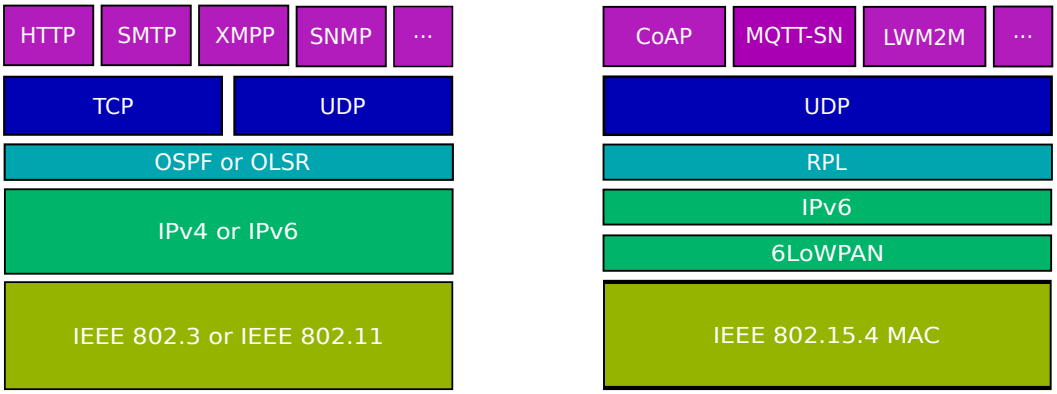


Figure 2.2: Comparing the traditional IP stack (left) to the new IoT IPv6 stack (right).

Access mechanisms determine which level of reliability, energy efficiency, and performance the link layer can provide.

IPv6 as the Narrow Waist

As discussed in Section 1.1 standardization is mandatory for the breakthrough of the IoT on a global-scale. The current Internet would not have become a success story without well-defined and specified protocols on all layers, neither would have UNIX or Linux without the standards like ANSI C or POSIX.

Consequently, nowadays nearly all communication in the Internet is IP-based. IP networks and architectures have proven to perform well over the last more than 30 years. However, for IoT IPv4 is not a viable choice, because of the limited number of available addresses and less flexible protocol header. Hence, in IoT scenarios IPv6 has become the default choice to seamlessly integrate LLNs into the Internet. A comparison of such an IPv6-based IoT stack to the traditional Internet stack can be seen in Figure 2.2. In accordance with this observation, the following sections will survey the IPv6 protocol suite for IoT systems. The impact on energy consumption and reliability of the presented protocols and mechanisms is particularly highlighted.

2.3.1 Medium Access

This section reviews some mechanisms that can be deployed on the lower layers of the network stack, namely the MAC layer, to improve reliability and availability. In contrast to the following sections, MAC protocol categories are reviewed here rather than concrete protocols and prominent representatives in the IoT are cited instead. A brief survey over these different categories is presented as well

as a review of their pros and cons with respect to complexity, performance, and robustness.

CSMA

In Carrier Sense Multiple Access (CSMA) a transceiver first *senses* the (shared) medium, before it starts a transmission. In CSMA/CA and CSMA/CD it additionally applies a backoff algorithm after a collision before it tries sending again. Probably still the most common type of MAC protocols in many WMN networks are based on CSMA. Well-known and widely deployed standards that use CSMA are IEEE 802.11 [IEEE802.11, 2012] and IEEE 802.15.4 [IEEE802.15.4, 2011]. The reason for the popularity of CSMA approaches is two-fold: **(i)** its fairly low complexity and **(ii)** its good performance with respect to latency and throughput in low or medium traffic load scenarios.

However, pure CSMA has fundamental drawbacks when it comes to reliability: CSMA gives no guarantees at all. Accessing the medium may take an indefinite time or fail completely in high traffic cases.

Considering its energy efficiency, literature and actual implementations propose some optimizations. In the basic variant CSMA requires a node to put its transceiver into listening mode whenever it does not transmit itself. This leads to a high energy consumption when the transceiver is in RX model while no transmissions for this host are transmitted, the so-called *idle listening*. As a solution to reduce *idle listening* phases, Radio Duty Cycling (RDC) approaches like ContikiMAC can be used [Dunkels, 2011]. With this approach a transceiver can be switched off for most of the time. As a consequence, the sender has to repeat the transmission until the recipient has received it. The transmission repetition can be stopped after a link layer acknowledgement is received for unicast transmissions. For broadcast transmissions it has to be repeated with respect to the RDC rate. Hence, sending of packets becomes energy-wise more expensive, especially for broadcast transmissions.

FH-CDMA

In Frequency-Hopping Code Division Multiple Access (FH-CDMA) a transceiver switches quickly between frequency channels following a known sequence. Deploying a FH-CDMA protocol is one approach to increase the robustness of medium access. Well-known and widely used representative for FH-CDMA protocols are Bluetooth (originally specified as IEEE 802.15.1) and Bluetooth Low Energy (BLE) [Bluetooth SIG, 2016]. FH-CDMA protocols are robust against external interference and multipath fading. They also allow to deploy multiple spatially overlapping networks. An additional advantage of FH-CDMA protocols is their

potential for adaptivity. In case that particular frequencies are (temporarily) unavailable due to interference or jamming, FH-CDMA protocols can avoid (*blacklist*) the affected channels.

In contrast to CSMA, FH-CDMA protocols require coordination between the devices. In order to switch synchronously between channels, a frequency hopping scheme needs to be deployed. Since all nodes in the network use multiple channels, the full frequency spectrum is not available for co-located networks, leading to a suboptimal spectrum usage. Channel assignment algorithms and FDMA MAC protocols can serve to make better use of the available frequency bandwidth in order to mitigate internal as well as external interferences [Juraschek et al., 2013]. Collisions in FH-CDMA-based networks are still possible.

TDMA

In Time Division Multiple Access (TDMA) time is divided into different slots and each slot is assigned to a particular link or node. With a proper assignment TDMA protocols increase the robustness of the MAC layer. Cellular networks deploy TDMA protocols as part of the GSM standard [GSM, 2016]. Also accessing the medium in a fully coordinated and scheduled manner, collisions and intra-network interference can be completely eliminated. Thereby TDMA enables deterministic behavior and timings, even at high traffic load.

As for FH-CDMA protocols, TDMA requires scheduling and clock synchronization. A further drawback is the comparably low throughput and high latency in low traffic scenarios. This can be mitigated by dynamically adapting the reservations. Some of the most popular representatives for TDMA protocols for LLNs like **(i)** *Z-MAC* [Rhee et al., 2008], **(ii)** *TRAMA* [Rajendran et al., 2006], and **(iii)** *BMA* [Li and Lazarou, 2004] additionally combine TDMA with CSMA.

Finally, it has to be mentioned that finding an optimal schedule for a transmission schedule is *NP*-hard [Ramanathan, 1997].

TSCH

In order to achieve both, a low energy consumption and a high reliability, combining TDMA and FH-CDMA promises best results. While TDMA can be used to rule out collisions inside the network and accomplish deterministic behavior, FH-CDMA makes the network more robust against external interference and multipath fading. Combining frequency hopping with time-slotting also increases the space for the scheduling algorithm. As a consequence, several MAC protocols implementing this approach in industrial IoT context evolved in this direction over the last decade.

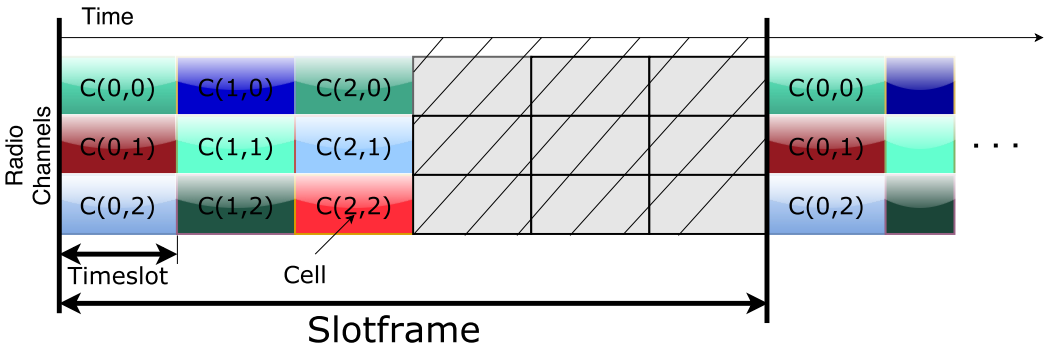


Figure 2.3: Simplified representation of a TSCH Schedule, with time slots represented horizontally, and channel offsets represented vertically.

The first step in this direction was the invention of the *Time Synchronized Mesh Protocol* (TSMP) [Pister and Doherty, 2008] in 2006. At the core of a TSMP network lies the *Time Synchronized Channel Hopping* (TSCH) technology: nodes tightly synchronize to reduce energy consumption, and exploit frequency diversity (in addition to special diversity) to obtain wire-like 99.999 % reliability (as reported in [Doherty et al., 2007], after running a 44-node network for 26 days and observing 2 millions of packets out of which only 7 have been lost).

The success of this approach resulted in standardization efforts such as WirelessHART [HART Communication Foundation, 2008] and IEEE 802.15.4e [IEEE802.15.4e, 2012]. TSCH is a MAC protocol which divides time into slices of fixed length that are grouped in a slotframe (compare Figure 2.3). Nodes are synchronized and share the notion of a slotframe which repeats over time. Channel hopping is achieved by sending successive packets on different frequencies. The channel hopping sequence is fixed and known by all nodes. In any given cell (a timeslot at a particular frequency), a node may transmit, listen, or sleep. The scheduler builds the communication schedule (i.e. allocates communication cells in the slotframe to different pairs of communicating nodes) in order to satisfy the bandwidth, latency, and reliability requirements of the applications. It is the task of the scheduler to keep the number of scheduled cells (in which a mote either transmits of listens) to a minimum in order not to waste energy.

Summary

The most prominent protocols on this layer are **(i)** basic IEEE 802.15.4 with CSMA providing link layer retransmissions for improved reliability and support for RDC to reduce energy consumption, **(ii)** BLE with a FH-CDMA MAC providing improved robustness and adaptivity to improve reliability, and **(iii)** IEEE 802.15.4e

with TSCH providing reservation-based medium access and channel hopping for a high degree of reliability and determinism.

In this thesis I will focus on the widely used IEEE 802.15.4 link layer and its default MAC protocol using CSMA. Some parts of the thesis will also consider the TSCH MAC amendment.

2.3.2 Network

IP represents the narrow waist of today's Internet and will represent the narrow waist for the IoT in the foreseeable future, too. In order to adapt IPv6 to the particular properties of LLNs additional mechanisms and optimizations are required.

6LoWPAN

As observed, IPv6 is the default narrow waist for the standard IoT stack. However, its required minimum Maximum Transmission Unit (MTU) of 1,280 bytes, rather large headers, and a chatty ND mechanism are prohibitive for direct use on top of IoT link layers. As a consequence the IETF developed several specifications to transport IPv6 over such constrained link layers. In 2007 the first of these specifications was released, as IPv6 over IEEE 802.15.4 networks (6LoWPAN) [Montenegro et al., 2007].

This original RFC was updated and extended by RFC6282 [Hui and Thubert, 2011], describing stateless header compression mechanisms, and RFC6775 [Shelby et al., 2012], which defines a variant of IPv6 ND that works in a more reactive way and requires less broadcast traffic. RFC6282 also describes mechanisms to compress next headers, e.g. for UDP, and RFC7400 [Bormann, 2014] describes a generic mechanism to compress additional headers of encapsulated protocols. Over the following years, more of the so-called *IP-over-foo* specifications for other prominent LLN link layers followed, such as *IPv6 over BLE*, *IPv6 over ITU-T G.9959 Networks* (6lowpanz), *IPv6 over MS/TP Networks* (6lobac), *IPv6 over Near Field Communication (NFC)*, or *IPv6 over DECT Ultra Low Energy* [6lo, 2016]. Most of these specifications make use of the IPv6 ND optimization [Shelby et al., 2012] and IPv6 Header Compression (HC) [Hui and Thubert, 2011] which where originally developed for IPv6 over IEEE 802.15.4. Several of these specifications define also how to fragment IP datagrams over link layers that do not support the minimum MTU for IPv6 of 1280 bytes.

6TiSCH

The TSCH MAC protocol as presented in Section 2.3.1 has become the *de-facto* standard for industrial IoT applications. However, legacy TSCH solutions do not sup-

port IP as an upper layer. As a consequence, the IETF chartered the IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) WG to develop a set of specifications for an IPv6-stack on top of TSCH.

In a 6TiSCH network a transmission schedule can be computed in order to achieve, for example, flow-ordered transmissions to fulfill packed deadlines requested by the application. This schedule can be computed either by a central instance, e.g. a Path Computation Element (PCE), or in a distributed manner.

The advantage of a 6TiSCH network over a regular 6LoWPAN network, using a CSMA MAC layer is obviously its more deterministic behavior and increased robustness. The price one has to pay for these benefits is the increased complexity of the architecture and decreased flexibility to changes in the network, in particular mobile nodes are difficult to handle.

The particular needs of the reservation-based MAC layer require some additional protocols. Directly on top of IEEE 802.15.4e the 6TiSCH Operation Sublayer (6top) protocol is responsible for setting up the TSCH schedule. 6top also creates and maintains an abstract neighbor table with additional information and statistics about the links to its neighbors It acts as an interface to allocate and deallocate schedule reservations to the upper layers.

Necessary synchronization traffic can be piggybacked with periodic Internet Control Message Protocol (ICMP) traffic required for RPL (see next section) and/or IP ND and does not necessarily cause additional communication overhead. In order to do so RPL or ND needs to be coupled with 6top. The RPL tree structure can be also leveraged for efficient distributed scheduling where a node negotiates cell reservations only with its parent.

Summary
Over the last ten years the IETF has released various specifications to support the transport of IPv6 over several LLN technologies. The most relevant specifications in scope of this thesis are 6LoWPAN and 6TiSCH. The main mechanisms described in these specifications define how packets can be compressed and ND can be conducted in a less chatty manner in order to reduce the energy consumption.

2.3.3 Routing

There are several mechanisms how routing protocols for the IoT can improve on energy efficiency and reliability:

- **Routing Metrics**
  Metrics that prefer more stable links (e.g., based on ETX[3] or RSSI) over a lower hop count can increase reliability and decrease the required retransmissions and consequently reduce energy consumption.
- **Repair Mechanisms**
  The ability to repair broken routes, in case of failing nodes or links, are also important for the reliability of the network.
- **Multipath Routes**
  If a routing protocol provide multiple paths to the destination, the network becomes more robust against failures.

These mechanisms can also help to improve the Quality of Services (QoS) or Quality of Experience (QoE) in WMNs [25].

RPL

Since none of the existing routing protocols was designed for the particular requirements of LLNs, the IETF chartered the ROLL WG [roll, 2016] which published several documents to define the particular requirements on the routing protocol for LLNs [Dohler et al., 2009, Pister et al., 2009, Brandt et al., 2010, Martocci et al., 2010]. As a result of these derived requirements the WG developed the specification for RPL. RPL follows a *one-fits-most* approach. In order to address the varying requirements, the selected routing metric is plugged in as a so-called *objective function*.

The routing protocol designed by the IETF for constrained environments is called RPL [Winter et al., 2012]. As a routing protocol, RPL has to exchange messages periodically and keep routing states for each destination.

It supports different traffic patterns:

- **Many-to-one communication**
  In this pattern nodes in the network primarily send data towards the root node of the RPL network. This root node is typical the data sink or a gateway, e.g. the 6LoWPAN Border Router (6lbr).
- **One-to-many communication**
  In this pattern the root node of a RPL network primarily sends data towards the other nodes, either leaf nodes or other routing nodes. The communication can be either unicast or multicast.
- **Any-to-any communication**
  In this pattern any pair of nodes in the RPL network may communicate.

[3] Expected Transmission Count (ETX) is a routing metric that describes the expected number of transmissions which are necessary to be received.

Improved support for this traffic pattern is specified in an separate document [Martocci et al., 2013].

While *many-to-one* communication is supported in all configurations—each node always configures the *default* route towards its parent node—other patterns require support for so-called *downward routes*. RPL offers two basic Mode of Operations (MOPs) to support these downward routes: **(i)** *storing* and **(ii)** *non-storing* mode. In *storing* mode, each node maintains a Forwarding Information Base (FIB) entry for all the destinations in its subtree. Hence, packets destined from one (non-root) node towards any other (non-root) node need to travel up the tree only until they reach the first common ancestor node. However, this MOP requires more memory resources per node. Since any subtree can become arbitrary large without further restrictions (e.g. topology control mechanisms), this MOP is sometimes considered to be "broken" [ROLL Mail Archive, 2016]. In *non-storing* mode, packets that need to travel down the tree are always required to traverse the RPL root node. The root node will then inject an IPv6 source routing header. Hence, the general tradeoff between *storing* and *non-storing* mode is between bigger memory requirements and bigger packet sizes.

With respect to energy-consumption, low control traffic is an important requirement for a routing protocol in LLNs. RPL was analyzed with Powertrace [Dunkels et al., 2011]. The analysis concluded that the ICMP-based control traffic decreases over time if a rather stable network is assumed. In order to tune the control traffic of RPL, the parameters of *Trickle* [Levis et al., 2011]—which is used to send out the *DODAG Information Objects* can be adapted. However, a trade-off exists between its energy efficiency and its performance, like network convergence time and robustness.

RFC6551 specifies a *Node Energy* object. This provides information related to node energy, e.g., the energy source type or an estimation of the remaining energy [Vasseur et al., 2012]. This can be used for a routing metric aiming for energy efficiency, to balance between energy consumption and network performance.

MANET protocols: Open Link State Routing (OLSR) and Ad-hoc On-demand Distance Vector routing (AODV)

Over the last two decades various routing protocols for WMNs have been developed. In general, theses protocols can be categorized into *proactive*, *reactive*, and *hybrid* routing protocols [Campista et al., 2008]. Proactive routing protocols—similar to the ones in traditional wired networks—search and store routes to any destination in advance. Reactive protocols request routes for a certain destination only on demand. Hybrid protocols combine both approaches. The responsible IETF WG

specified OLSR [Clausen and Jacquet, 2003, Herberg et al., 2014] (in two versions) as proactive approaches and AODV [Perkins et al., 2003] as reactive protocol.

OLSR The Optimized Link State Routing Protocol is a proactive mesh-routing protocol. That means it will try to calculate and maintain the shortest route to any node in the network, as soon as it learns about its existence. It is, as the name suggests, also a link-state protocol, meaning that it will keep track of the entire way towards a node in the network. The protocol operates above the transport layer (mesh over) and communicates using broadcast UDP packets.

AODV AODV was designed with a special focus on mobile nodes in ad hoc networks. This should be achieved by fast adaptations to changing link conditions. When a link fails, AODV notifies the affected set of nodes, to allow them to invalidate the routes using the failing link. It furthermore aims for low processing complexity, a small memory overhead, and little traffic. Destination sequence numbers are used to ensure that the routing graph is free of loops at all times and avoiding *count to infinity* problems that may occur in classical distance vector protocols. In AODV nodes do not need to maintain routes to destinations that are not addressed. If a node wants to send a packet to a yet unknown destination it sends out route requests (RREQ) which gets forwarded until it reaches the destination or another node that has information about how to reach this destination. In this case route reply (RREP) is sent back to the requesting node.

Summary
The main routing protocols for LLNs are RPL in various MOPs and the MANET protocols OLSR(v2) and AODV(v2). RPL supports multiple routing metrics and traffic patterns in order to fulfill energy consumption and reliability criteria and is designed to operate with strict memory constraints. The MANET protocols are designed in a less memory-efficient way, but provide mechanisms to improve the performance or deal with mobile nodes.

For this thesis, I focus on RPL as default LLN routing protocol.

2.3.4 Transport Layer

Transport layer protocols can offer the following services to the upper layers, most prominent the application layer:

- **In-order delivery.**
  It is ensured that datagrams (or segments) are delivered to the upper layers

in the same ordered they were sent. This requires typically some kind of sequence numbers and buffering at the receiving side.

- **End-to-end acknowledgements.**
  By requesting a confirmation from the recipient, it is ensured that sent data is delivered. If no confirmations is received after a certain time, the Protocol Data Unit (PDU) is retransmitted. As a consequence, packet loss in both directions and unexpected high latency can cause a retransmission of the transport layer PDU.
- **Flow-control and congestion avoidance.**
  Overloading both, the receiver and the lower layers must be avoided. A flow-control mechanism typically uses some RTS/CTS or *sliding window* mechanism to avoid overloading.
- **Segmentation and reassembly.**
  If a data packet is larger than the MTU supported by the network, the transport layer can offer to split packets into smaller units, so-called segments, and reassemble them on reception. This segmentation service may also include mechanisms to detect and request missing segments.

TCP

TCP is a transport layer protocol that was introduced in the early days of packet switched networks and is one of the core Internet protocols and widely used in nowadays networking applications. For IoT however it is often considered inappropriate for several reasons:

- **Counterproductive Congestion Control**
  TCP responds to lost segments by invoking congestion control mechanisms. However, packet loss in WMNs is often caused by other reasons than congestion.
- **Retransmission Timeout**
  Due to the lossy nature of LLNs and link layer retransmissions the Retransmission Timeout (RTO) for TCP is difficult to estimate.
- **Complexity**
  The rather complex protocol and requirement for segment buffers do not fit the constrained resources of low-power IoT devices.

However, a properly configured TCP implementation using appropriate mechanisms for congestion control and RTO estimation could make TCP useful for LLNs. An according proposal has been published in [Gomez and Crowcroft, 2016]. Consequently, implementations of basic TCP versions exist for IoT OSs such as RIOT or Contiki [17][Dunkels et al., 2004].

CoAP

The reader may wonder why CoAP, which is obviously an application layer protocol, is listed additionally here as a transport layer mechanism. The motivation for this is that the CoAP specifications include many mechanisms that can be rather attributed to the transport layer. This is a result of the consideration that TCP may be too complex for LLNs and therefore User Datagram Protocol (UDP) is preferable. As a consequence, mechanisms for a reliable transport are outsourced to the upper layer, e.g. CoAP. Note, that the content-aware aspects of CoAP are described in Section 2.3.5.

The following transport layer services are provided by CoAP:

- **end-to-end acknowledgements** by *confirmable* messages as specified in [Shelby et al., 2014].
- **congestion control** by a simple or a simple advanced congestion control mechanism as specified in [Shelby et al., 2014, Bormann et al., 2016].
- **improved RTO estimation** by the CoAP Simple Congestion Control/Advanced (CoCOA) RTO estimation as specified in [Bormann et al., 2016].

Summary

The main protocols on this layer are **(i)** CoAP providing end-to-end acknowledgements and congestion control and **(ii)** TCP providing in-order delivery, end-to-end acknowledgements, flow-control, and congestion avoidance.

In this thesis I focus on UDP on the transport layer, complemented by additional CoAP mechanisms, targeting transport layer services.

2.3.5 Content Aware

This section will survey protocols that have a notion of the content they are transporting, i.e. which are located at the application layer or above. As a consequence of the wide range of use cases in the IoT, the focus is on the most prominent representatives.

The main mechanisms in this layer with respect to energy efficiency and reliability are:

- **Traffic Flow Optimization**
  Based on the knowledge about the importance of content, mechanisms can prioritize certain traffic flows or aggregate content in order to reduce retransmissions.
- **Selection of appropriate Lower Layer Services**
  Content aware mechanisms select the proper services from lower layers, e.g.,

transport layer or routing algorithms, to provide the proper QoS levels with respect to the relevance of the content.

- **Resource Discovery**
  Energy efficient mechanisms to discover resources (services or data) provided by other hosts in the LLN are provided.
- **Publish-Subscribe Models**
  Content aware mechanisms provide energy efficient and reliable models to publish data and subscribe for certain content.

Due to the constraints and particular requirements in IoT, protocols in this area cannot always be mapped to the well-known ISO/OSI protocol stack. Consequently, some of the protocols that are presented in this section comprise typical application layer functionalities as well as mechanisms that are usually rather part of the transport layer[4]. This section will concentrate on the content aware aspects of the discussed protocols. Transport layer aspects are discussed in one of the following sections.

CoAP

CoAP is an application protocol, developed by the IETF Constrained RESTful Environments (CoRE) WG, that implements the *Representational State Transfer (REST)* paradigm, in order to connect LLN services to the World Wide Web (WWW) [Shelby et al., 2014]. In this sense, it acts as a replacement for HTTP, addressing the particular constraints of LLNs by using binary headers and reducing control traffic. The header has a fixed length of only four bytes, followed by binary options.

The design of CoAP also addresses energy efficiency. The observe mode can be used as an extension to reduce regular and frequent resource queries [Hartke, 2015]. Hereby the requester registers for a certain resource and will get automatically updated by the resource provider whenever the resource is updated. In this way the request-response round trip time is decreased without harming the availability of the requested resource. From an energy saving point of view, this additionally allows a server to keep sleeping until a corresponding resource has been updated and needs to be transmitted.

Furthermore, the deployment of CoAP proxies can help to reduce energy consumption by caching the resources of sleeping CoAP servers. The proxy can either respond to client requests itself if the timestamp of the resource is recent enough or attempt to request a newer version from the sleeping server.

[4] In the presented IPv6 IoT stack UDP is used as the base transport layer.

MQTT and MQTT-SN

MQ Telemetry Transport (MQTT) is a publish-subscribe messaging protocol [Banks and Gupta, 2015]. A producer *publishes* a message on a particular subject, a so-called *topic*. A consumer *subscribes* for messages for this a topic. Subscriptions in MQTT always belong to a topic. Topics in MQTT have maximum length of 64 kB, are hierarchically structured using a "/" as separator.

MQTT also includes support for detecting failed connections and introduces keep alive signalling between clients and server. MQTT provides three, so-called, QoS-levels with respect to delivery. These levels guarantee that a message is delivered **(i)** at most once (QoS 0), **(ii)** at least once (QoS 1), **(iii)** or exactly once (QoS 2) to the receiver. If *at most once* is specified, no retransmissions are used. This is the default QoS level and is used if occasional message loss can be tolerated. If *at least once* is specified, a message identifier is specified in the message header and the receiver can acknowledge the packet using this message identifier. A sender that does not receive an expected acknowledgement, it sets a duplicate bit in the message header and retransmits the message. This QoS level is used when guaranteed message delivery is required and duplicates can be tolerated. If *exactly once* is specified, at least two message exchanges are necessary, After the sender has received the acknowledgement from the receiver, it tells the receiver that it can complete processing the message and waits for an acknowledgement for this signalling. Consequently, this is the most reliable QoS level, but introduces some overhead and is the slowest variant.

MQTT requires TCP or any other transport service that provides ordered, lossless, bi-direction connections. In order to target resource-constrained devices that may not provide TCP MQTT for Sensor Networks (MQTT-SN) was introduced [Stanford-Clark and Truong, 2014]. The main differences to MQTT are that it allows UDP at the transport layer, that it only supports one concurrent connection to a gateway, and that the number of topics a client can subscribe to are limited.

DDS

Data Distribution Service (DDS) is a middleware standard released by the OMG [OMG, 2014]. OMG is responsible for widely accepted standards like UML, MDA, CORBA, BPNP, etc. DDS provides a data-centric, publish-subscribe API with an extensive set of QoS policies. As an OMG standard, DDS is vendor and platform independent, with different commercial and open-source implementations available. Additional specifications cover aspects such as interoperability, security, and API mappings to programming languages.

The feature set of DDS satisfies the requirements of many IoT scenarios and applications, where data have to be collected and transported to interested parties.

OMG is promoting DDS as "the standard" for industrial IoT scenarios, within the Industrial Internet Consortium (IIC) [IIC, 2016]. DDS implementations typically target embedded systems and resource usage can be controlled directly through the API. However, low-end IoT devices, as they are the focus of this document, are usually too constrained for these implementations. Moreover, the official wire-protocol is too large for LLNs.

To incorporate the resource restrictions of small systems, Sensor networks DDS (sDDS) was proposed in [Beckmann and Dedi, 2015]. Following a model driven software development process, it provides a DDS framework tailoring the middleware functionality to the requirements of the application and capabilities of the hardware. The main design goals were s small memory footprint, fine-granular resource consumption configuration, low protocol overhead, and good portability. sDDS utilise existing system software layers and network transport facilities of target platforms. Currently two common OSs for IoT are supported: Contiki and RIOT. Furthermore, sDDS can run as a Linux process on the 6lbr.

Summary
The main protocols on this layer are **(i)** CoAP providing services like proxies and observe mode to reduce energy consumption, **(ii)** MQTT(-SN) providing a publish-subscribe model and three QoS levels to support reliability, and **(iii)** DDS also providing a publish-subscribe model along with flow-control, and retransmissions to improve reliability.

Since the REST paradigm is widely used in the traditional Internet and consequently also intended to be applied for many IoT applications, I refer to CoAP as a default solution for the rest (sic!) of the thesis.

2.4 Auxiliary Mechanisms & Frameworks LLNs

The previous sections described the core protocols of an IPv6-based IoT network stack. Additional to these central protocols between link and application layer, this section will present some additional mechanisms. These mechanisms may provide services to particular protocols in the stack. This can be either mandatory if these protocols have certain requirements, such as synchronization between the nodes or some kind of negotiation, or in order to facilitate certain applications.

2.4.1 Security

Mechanisms to ensure secure communication can be deployed on several layers of the network stack. Usually, it is neither necessary nor reasonable to deploy security

measurements on all layers, but rather pick the proper mechanisms for a certain subset.

On the one hand, for instance link layer security supports in-network processing, passive participation and local broadcast to save traffic and reduce energy. End-to-end security, on the other hand, provide a secure communication to the application, but may not be able to support these operations without, e.g. a proxy.

Cryptographic functions are particular challenges for resource constrained devices of LLNs deployments. An analysis of implementation considerations and the general feasibility of these mechanisms is presented in [Sethi et al., 2016]. They demonstrated that using the Relic cryptography toolkit a secure prototype application can be implemented with about 51 kB ROM and 5 kB RAM usage. The generation of a key pair, sending a registration message, and getting the acknowledgement takes about 2 s, signing the hash of a message and sending the update takes about 1 s. In [Gura et al., 2004] the authors compare elliptic curve cryptography and RSA on 8-bit MCUs. They present that a RSA-private key operation with a key length of 2048 bytes[5] takes about 83 s. Corresponding Elliptic Curve Cryptography (ECC) operations with parameters recommended by Standards for Efficient Cryptography Group (SECG) took between 810 ms and 5.37 s.

This section will briefly introduce some prominent security mechanisms per layer in an IP-based IoT network. A more detailed security analysis for IoT scenarios is out of scope for this thesis.

IEEE 802.15.4 Security Sublayer The IEEE 802.15.4 security sublayer optionally secures the wireless link and allows upper layers as at least as securely as they would do over a wired connection. Security is guaranteed per frame, where *ACK* frames are not secured. Secured frames contain an additional Auxiliary Security Header (ASH) and optionally a Message Integrity Code (MIC) for authentication.

Three different security suites exist: **(i)** the *CTR* level provides confidentiality, **(ii)** the *CBC-MAC* level provides authentication and replay detection, and **(iii)** the *CCM* level provides authentication and confidentiality. AES block ciphers with a fixed block size of 128 bit and a key length of 128 bit is used. Key distribution mechanisms are not part of the IEEE 802.15.4 specification. An implementation on TinyOS requires about 6 kB of ROM and 4 kB of RAM with hardware-based cryptography and about 9 kB of RAM and 5 kB of RAM with software-base cryptography [Daidone et al., 2014].

IPsec Internet Protocol Security (IPsec) as specified in RFC4301 provides authentication and privacy for IPv6 [Kent and Seo, 2005]. Actually, it defines a set of mech-

[5] Shorter RSA keys must be considered insecure.

anisms: **(i)** the security protocols *Authentication Header (AH)* and *Encapsulating Security Payload (ESP)*, **(ii)** algorithms for *authentication and encryption*, **(iii)** *key exchange* mechanisms, and **(iv)** *security associations (SAs)*. A SA defines the security level of an IP flow. Establishing SAs can be done using PSKs or using the *Internet Key Exchange (IKE)* protocol. IKE requires asymmetric cryptography, which has been considered too heavy weight for constrained devices for a long time, but may be feasible by using ECC. AH provides connectionless integrity, data origin authentication, and protection against replays by adding a *Message Authentication Code (MAC)* to the IP header. ESP provides origin authenticity, integrity, and confidentiality protection. It is used to encrypt the payload, but does not secure the IP header. The following cryptographic algorithms are defined for the use with IPsec: **(i)** *HMAC-SHA1/SHA2* for integrity protection and authenticity. **(ii)** *TripleDES (3DES)-CBC* for confidentiality[6]. **(iii)** *AES-CBC* for confidentiality. **(iv)** *AES-GCM* providing confidentiality and authentication together efficiently. An implementation of AH and ESP on Contiki requires between 11 kB and 14.5 kB of ROM and between 0.8 kB and 1.4 kB of RAM (depending on the used cryptography) [Raza et al., 2011].

DTLS Securing the transport layer in daily Internet communication is usually implemented by using Transport Layer Security (TLS) [Dierks and Rescorla, 2008]. However, it requires a stream-oriented transport layer like TCP which is often not available in LLNs (compare Section 2.3.4). Moreover, application layer protocols in this context are rather built upon UDP. Datagram Transport Layer Security (DTLS) was proposed as an alternative to TLS to provide a secured transport layer over an unreliable, datagram oriented transport layer, such as UDP [Rescorla and Modadugu, 2012]. It inherits some characteristics from TLS. It consists of two layers, the *Record Layer* that is the basis for these four mechanisms: **(i)** *Handshake*, **(ii)** *ChangeCipherSpec*, **(iii)** *Alert Protocol*, and **(iv)** *Application Layer Protocol*. The initial handshake serves to authenticate the server (and optionally the client), e.g. using Public Key Infrastructure (PKI), negotiating the algorithm and establishing keys.. ChangeCipherSpec is used by the client during this handshake to indicate changing to the negotiated cipher suite. The Alert Protocol is used to communicate error messages between the peers. The header of the carrier protocol, the Record Layer, inserts a header containing the content type and fragment fields. An implementation of DTLS on RIOT and Contiki requires about 17 kB of ROM and about 4 kB of RAM [Raza et al., 2013]..

[6]Since TripleDES is very slow when implemented in software, it should not be used in LLNs without hardware acceleration

OSCoAP In order to provide security mechanisms on the application layer, Object Security CoAP (OSCoAP) was proposed. OSCoAP uses the Concise Binary Object Representation (CBOR) Object Signing and Encryption (COSE) format to provide end-to-end encryption, integrity, and replay protection for CoAP (header, options, and payload) [Selander et al., 2016]. A new CoAP option, the *object security option* is introduced to signal the use of OSCoAP. The original CoAP message is secured by inserting a COSE object that comprises the static part of the CoAP header, certain options and the payload. The encrypted parts are removed from the message. Some options like `Uri-Host` or `Proxy-*` are not protected or encrypted. Compared with DTLS, OSCoAP provides end-to-end security even if a proxy—that requires the DTLS connection to be terminated—is involved. OSCoAP requires Advanced Encryption Standard (AES)-CCM-64-64-128 with a key length of 128 bit. An implementation of OSCoAP on Contiki requires about 31 kB of ROM and 1 kB of RAM [Brorsson and Gunnarsson, 2016][7].

Summary
The main objectives of confidentiality, integrity, authentication, and authenticity can be achieved on different layers. While applying security measurements on all layers at the same time is neither efficient nor sensible in LLNs, eligible mechanisms and corresponding implementations for each layer are available. Selecting the appropriate and properly configured security mechanism for a particular IoT use case is up to the application developer. In this thesis I will not focus on security, but will assume that the mentioned security mechanisms can be complementary deployed.

2.4.2 Network Management

A particular class of application layer protocols are network management protocols. Network management protocols like Simple Network Management Protocol (SNMP) are required for configuring and monitoring networking properties. A network management protocol can also be used to disseminate resource reservations, e.g. transmission schedules for a network with a reservation-based MAC protocol. The management data used for SNMP and traditional Internet deployments are provided by Management Information Base (MIB) objects. MIB objects are defined by a subset of ASN.1. Lately, the IETF encouraged protocol specification developers to use NETCONF and YANG as a replacement for SNMP and MIB [Shafer, 2011]. However, neither of the traditional approaches is suited for the requirements of LLNs, as they require TCP at the transport layer and use space hungry XML for-

[7] Additionally to the plain CoAP implementation

mat. The IETF defined the requirements for an network management approach for LLNs in RFC7547 [Ersue et al., 2015].

CoMI Consequently, a CoAP based management protocol was specified as CoAP Management Interface (CoMI) [van der Stok and Bierman, 2016]. CoMI is designed to facilitate automatic management of large number of nodes and provide reduced complexity and runtime resources. Similar to the RESTCONF [Bierman et al., 2016] specification, that describes an HTTP-based protocol to configure YANG objects, but with CoAP instead of HTTP. It uses CoAP's `GET`, `PUT`, `PATCH`, `POST`, and `DELETE`. Payload is encoded using CBOR. Using YANG objects promotes interoperability and allows access to a large amount of existing YANG and even MIB specifications if the server converts the modules to YANG according to RFC6643 [Schoenwaelder, 2012]. In order to represent management objects in an efficient manner, a CBOR mapping for YANG objects was proposed [Veillette et al., 2016].

LWM2M Another protocol targeting network management for LLNs is specified by the OMA as OMA Lightweight M2M (LWM2M) [OMA LwM2M, 2016]. It is a replacement of the OMA *Device Management (DM)* protocol for constrained devices and decines a client-server architecture. Like CoMI LWM2M is built on top of CoAP and DTLS and uses CoAP's `GET`, `PUT`, `POST`, and `DELETE`. It supports device monitoring and configuration, server provisioning for bootstrapping, and software updates. Plain text is used for singular resources, while binary TLV or JavaScript Object Notation (JSON) for resource batches. Six normative objects are defined: **(i) LWM2M Server** providing data related to the server, initial access rights, and security related data, **(ii) Access Control** to check whether the server is allowed to perform an operation, **(iii) Device** describes device related information and a reboot/reset interface, **(iv) Connectivity Monitoring** enables network connectivity monitoring, **(v) Firmware** for firmware updates, and **(vi) Location** providing GPS information. Four interfaces between server and client are defined between the server and client: **(i) Bootstrap** to manage keying, access control, and client enrolment, **(ii) Device Discovery and Registration** to register clients and their objects, **(iii) Device Management and Service** to access objects and resources from the server, and **(iv) Information Reporting** for resource information notifications. A performance analysis in [Rao et al., 2015] reports that a LWM2M implementation requires about the same memory as CoAP implementation (< 9 kB of ROM and < 1 kB of RAM). Popular LWM2M implementations are provided as *Wakaama* (C) and *Leshan* (Java) by the *Eclipse* Foundation.

Summary
Several network management protocols are designed for the 6LoWPAN IoT stack, typically on top of CoAP. Both, prominent protocols CoMI and LWM2M provide SNMP-like functionality on top of a REST architecture using a compressed data representation format. CoMI may be used to configure network wide traffic flows and setup reservations providing a application interface to 6TiSCH's 6top. In this thesis I will not focus on network management, but will assume that the mentioned security mechanisms can be complementary deployed.

2.4.3 Clock Synchronization

For a coordinated medium access like TDMA or FH-CDMA a synchronized time base between the nodes is a requirement. However, traditional clock synchronization protocols like Network Time Protocol (NTP) or Precision Time Protocol (PTP) are not suitable for WMNs in general and LLNs in particular [Mills, 2006, IEEE1588, 2014]. Consequently, several clock synchronization protocols have been proposed particular tailored for the needs of WSNs and LLNs. One of the main goals of these protocols is to achieve the required level of synchronization between clocks with a minimum overhead in terms of packet exchange, i.e. the synchronization intervals should be extended as much as possible. In my diploma thesis I demonstrated that synchronization intervals can be extended up to three hours on a MSP430 platform, if maximum synchronization error of up to 6 ms can be tolerated and no drastic environmental changes (e.g. temperature) occur [Hahm, 2007]. This could be achieved for a TDMA protocol along a routing tree, by not only compensating the offset, but also computing the drift of each node's parent. Thus the clock of all nodes converged towards the clock of the routing tree's root node.

State of the art clock synchronization protocols are Glossy or PulseSync [Ferrari et al., 2011, Lenzen et al., 2015]. Other approaches, like the "Adaptive synchronization", target directly the use case of TSCH networks [Stanislowski et al., 2014].

PulseSync The central idea of the PulseSync algorithm is to distribute the clock values as fast as possible with a minimal number of transmissions. This enables PulseSync to quickly adapt to changes in clock drift, by, e.g., temperature changes or varying battery charges. Flooding the network with a *pulse* of new clock value information is done on a breadth-first search (BFS) tree. Clock drift compensation is used as well to minimize clock skews at all times. As for all modern LLN clock synchronization protocols an accurate MAC layer timestamping is used.

Glossy Glossy leverages simultaneous transmissions of the same packet as *constructive interference*. It leverages the broadcast nature of the medium and let nodes overhear from their neighbors. After reception a packet is immediately received.

Since all nodes in the same broadcast domain receive the packet at the same time, it is relayed at the same time, resulting constructive interference. A relay-counter is embedded in each packet, so that each receiver can infer how many times the packet was relayed. This information and the knowledge about the time between two transmissions, a node can compute the clock offset against the initiator.

Adaptive TimeSynch In this approach the knowledge about the TSCH parameters and architecture is leveraged. Each node in a TSCH network selects one neighbor as a time source neighbor. In a 6TiSCH network this is typically its RPL parent. At first a node stores the absolute slot number (ASN) during which the synchronization takes place. During the next synchronization it measures the offset and computes the offset. To achieve coordinated multi-hop synchronization, nodes coordinate the instants at which they synchronize, so that a node synchronizes right after its time source neighbor has. They do so by adding a field to all enhanced beacon and ACK packets, indicating their synchronization period.

Summary
The main protocols for this purpose are **(i)** Glossy leveraging constructive interference, **(ii)** PulseSync minimizing the number of transmissions and required time for network wide synchronization, and **(iii)** Adaptive TimeSynch leveraging the about the TSCH architecture to reduce complexity and overhead.

In this thesis we will assume that one of the presented clock synchronization protocols is in place when needed. We will also assume that this protocol is deployed in a secure and reliable way. For several security mechanisms a reliable system time is mandatory.

2.4.4 Link-Layer Transmission Scheduling

Every node in a network with reservation-based MAC protocols, like TDMA or TSCH maintains a schedule (which repeats in every slotframe). The following describes how such a schedule for a TSCH network looks like and how it built. This schedule can be represented as a matrix (timeslots as columns and channel offsets as rows) where cells can be reserved for receiving, sending, or broadcasting (shared cells). Reserving cells in a TSCH schedule can be done either by using a *node scheduling* algorithm or a *link scheduling* algorithm [Dezfouli et al., 2015]. A node scheduling algorithm ensures that each node's transmission timeslot does not conflict with any transmission timeslot of its 1-hop or 2-hop neighbors. This guarantees that a node's transmission can be received by each of its 1-hop neighbors. Hence, this is a suitable approach for broadcast transmission. A link scheduling algorithm guarantees that each transmission of any node to a specific neighbor

is receivable by the intended receiver. This receiver-oriented scheduling is suitable for unicast transmission. Both types of scheduling techniques serve different purposes and accordingly, a TSCH schedule may include both *broadcast cells* and *unicast cells*. Examples of node scheduling algorithms include NAMA (Node Activation Multiple Access), based on the Neighborhood-aware Contention Resolution (NCR) algorithm [Bao and Garcia-Luna-Aceves, 2001]. Examples of link scheduling algorithms include LAMA (Link Activation Multiple Access) which also uses NCR.

The cells for a transmission schedule in TSCH are either subject to *static reservations* or to *dynamic reservation*. Static reservation allocates the cells once in the beginning and keeps this schedule until the node leaves the network. In contrast, dynamic reservation allows a node to reserve cells only on demand, e.g. in response to the node's current traffic load. Cells may be added or removed to the schedule at any time. However, negotiating and modifying these reservations introduces additional overhead. Periodic information exchange—either between neighboring nodes or towards a central entity—is necessary to (i) update the information on each node's current neighborhood and schedule and (ii) either a negotiation protocol between neighboring nodes [Zhu and Corson, 2001] or traffic for requesting and assigning the schedule by a central entity such as PCE [Farrel et al., 2006], or TASA [Palattella et al., 2012]. In this context, DICSA provides a distributed and concurrent link scheduling algorithm that requires no specific assumption regarding the underlying network [Dezfouli et al., 2015]. DeTAS [Accettura et al., 2013] provides another distributed link scheduling algorithm specifically targeting 6TiSCH [Watteyne et al., 2015]. Tinka et al. proposed a simple scheduling mechanism for the TSCH MAC protocol that aims for full connectivity with a focus on mobile nodes and a dynamically changing neighborhood [Tinka et al., 2010].

Summary

Various reservation protocols and algorithms have been proposed for LLNs. They can be categorized into static and dynamic approaches and may work in a centralized or distributed manner. These algorithms can either perform link or node scheduling. The main interesting approaches in the scope of this thesis are NAMA, TASA, and DeTAS.

I will focus on this problem in Chapter 8.

2.4.5 Interoperability Frameworks

While the number of available protocols and configuration options as described in Section 2.3 accounts for the heterogeneity and wide number of use cases in the

IoT, it leaves some problems unresolved and poses some new questions. The selection of the right protocols and mechanisms, choosing the right configuration parameters, and automatic commissioning are among these challenges. Several commercial and open-source alliances worked on frameworks and specifications to facilitate the application in certain IoT use cases. This section will briefly present some of the most prominent examples. A detailed description of these frameworks is out of scope for this thesis.

Thread [Nest, 2016b] The *Thread* group lead by Nest Labs (a subsidiary of Google) defines stack based on the presented 6LoWPAN stack. Their specifications are not freely available and can only be accessed by members of the alliance paying a membership fee. Only white papers are available for the public which serves as a basis for this overview. The specifications target mainly home automation use cases and have a special focus on bootstrapping, network security, and service discovery. It comprises IEEE 802.15.4 including link layer security, IPv6 and 6LoWPAN, a distance vector routing protocol, and UDP with DTLS on top. While most of these protocols are a logical choice following the observations in Chapter 1 and 2, it is a bit surprising that the routing protocol is based on the old *RIPng* [Malkin and Minnear, 1997], rather than a protocol that was especially designed for MANET or LLNs. Up to 250 devices per network are supported. An open-source implementation by Nest is available online on GitHub [Nest, 2016a]. A certification program program for devices is planned.

IoTivity [OCF, 2016] IoTivity is an open-source framework developed and provided by the *Open Connectivity Foundation (OCF)* (the former Open Interconnect Consortium). OCF is lead by Samsung Electronics, Intel, Microsoft, Qualcomm, and Electrolux. It focusses on the upper layers of the stack and uses CoAP with DTLS as a transport. Upcoming versions should also support HTTP and other TCP-based protocols. Consequently, many different link layer technologies like BLE, IEEE 802.11, IEEE 802.15.4, or International Telecommunication Union (ITU) G.9959 are supported. It provides a set of services on various layers, e.g., service and node discovery, device management, or security mechanisms. An official open-source implementation for desktop and mobile OSs (like Android or Tizen) is available on the *IoTivity* website. An alternative open-source implementation is available in the Soletta project supporting IoT OSs like RIOT or Zephyr. A certification program is under development, but not yet publicly available.

IP for Smart Objects (IPSO) [IPSO, 2016] The IPSO Alliance is a non-profit organization that promotes IP deployment in the IoT. It manages an object registry that

comprises libraries and repositories to be used by worldwide standard definition organizations, special interest groups, open communities, and original equipment manufacturers. IPSO specifications are based on a RESTful interaction model and LWM2M is used for network management and object definitions. Currently three WGs exist:

- *Semantic WG* focussing on interoperability across different information models developed by IPSO and other organizations,
- *Protocol WG* focussing on the interaction with other protocols such as QUIC or MQTT, and
- *Marcom WG* focussing on consulting on major marketing, branding and communications strategy on behalf of Member Companies.

Several implementations of IPSO are available on GitHub. To the best of our knowledge, no certification is provided or planned.

Summary
These frameworks may play an important role for commercial solution because of unified APIs, certification possibilities, or simply the branding. However, for the focus of this thesis, these frameworks are not relevant, since they do not offer particular new solutions to address the major challenges as identified in Section 2.2.

2.5 Other Paradigms

2.5.1 The Silo Approach

In contrast to the approach based on the IP suite and IETF standards that was presented in Section 2.3, other concepts aim for so-called *silo* solutions. These silos are typically optimized for a certain application domain. Inside these silos a custom-tailored protocol stack is deployed inside the LLN, while special gateways ensure connectivity with the Internet (as depicted in Figure 2.4.

The main advantage of this approach is two-fold: **(i)** A single provider of the whole stack ensures interoperability between the layers, application and the network stack, and between devices. **(ii)** Since silos are often tailored for one particular use case and do not aim for interoperability with the rest of the world, they can often provide a higher degree of optimization with respect to their use case.

The main drawbacks of this approach are: **(i)** A particular gateway device for each silo solution is required. **(ii)** These gateways may introduce a *single point of failure*, since they can usually not be replaced by any other device in the network. **(iii)** Silo solutions are often steered by a single company or a commercial consortium, which in turn creates a high degree of vendor dependency. **(iv)** They of-

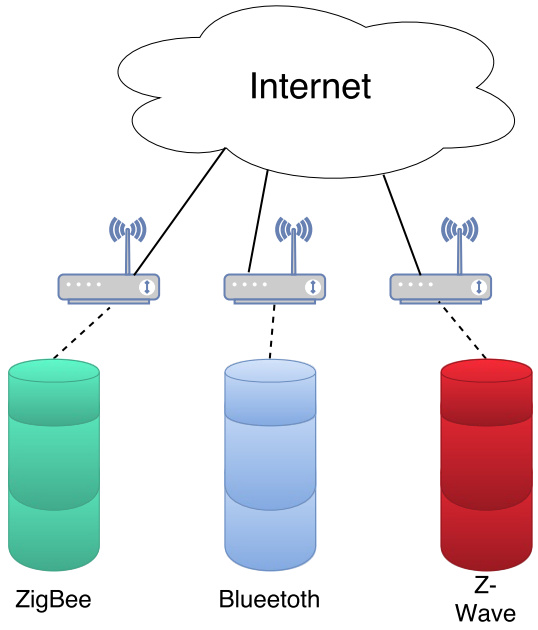


Figure 2.4: Different silo solutions connect to the Internet via dedicated gateway. Communication between nodes in each silo is only possible through the cloud.

ten base on closed specifications which are not open for everyone or prevent open source implementations by other means, e.g. requesting a certification fee.

This section will review some of these silo solutions.

ZigBee [ZigBee Alliance, 2016]
ZigBee specifies communication protocols that focus on Personal Area Networks (PANs). It is developed by the ZigBee Alliance since 2002. As a link layer IEEE 802.15.4 is deployed. (In fact, ZigBee and IEEE 802.15.4 are often confused.) Devices are classified into three categories: ZigBee Coordinators, ZigBee Routers, and ZigBee End Devices which loosely correspond to 6lbrs, 6LoWPAN Routers, and 6LoWPAN Nodes in 6LoWPAN networks. ZigBee networks may either use AODV or a tree-based routing protocol. A set of conventions describing type of messages, formats and so on are grouped in *profiles*. It defines multiple application profiles such as home automation, smart energy, health care, or commercial building automation. The ZigBee Device Object (ZDO) protocol is used for device management, key distribution, and policy management. The Application Support Sublayer (APS) provides an interface and control services between the application and the network layer functionalities. Some open-source implementations of the ZigBee stack exist, e.g. FreakZ [FreakZ, 2013], but licensing under GPL is prohibited

by the specification due to its licensing fees. Devices and implementations must be certified, paying a fee to the ZigBee Alliance.

Bluetooth [Bluetooth SIG, 2016]
Bluetooth define protocols focusing mainly on PANs. The Bluetooth protocol stack is developed by the Bluetooth Special Interest Group (SIG) since 1998. The physical layer operates in the 2.4 GHz Industrial, Scientific and Medical (ISM) band using a FH-CDMA MAC protocol with up to 50 Mbit/s. The Bluetooth architecture consists of the following protocols: **(i)** core protocols, **(ii)** cable replacement protocols, **(iii)** telephony protocols, and **(iv)** adopted protocols[8]. Each implementation of the Bluetooth stack has to implement at least the Link Management Protocol (LMP), the Logical Link Control and Adaptation Protocol (L2CAP), and the Service Discovery Protocol (SDP). L2CAP supports link layer fragmentation up to 64 kB, a streaming mode, retransmissions, and flow control. SDP uses Bluetooth profiles to perform service discovery. Since a star topology is mandatory in classical Bluetooth networks, no routing protocol is specified. Currently more than 30 profiles are defined, such as Audio/Video Remote Control Profile (AVRCP), Basic Imaging Profile (BIP), Basic Printing Profile (BPP), Hands-Free Profile (HFP), Human Interface Device Profile (HID) or the Serial Port Profile (SPP). Many different open-source implementations of the Bluetooth stack exist for different categories of OSs. Devices can be Bluetooth certified by the Bluetooth SIG.

Z-Wave [Z-Wave, 2016]
Another protocol stack mainly intended for home automation is called Z-Wave. It was developed by ZenSys which got acquired by Sigma Designs and is now promoted by the Z-Wave Alliance. The lower layers are specified by ITU as *G.9959*. The physical layer mainly in the sub-GHz and 2.4 GHz ISM bands and allows transmissions between 9.6 and 200 kbit/s. Z-Wave's MAC protocol is based on CSMA providing an optional retransmission mechanism. Two types of devices, controllers and slaves, are defined, where the controller polls or sends commands to the slaves, which reply to the controllers or execute the commands. A source routing approach is used with a maximum of four hops, which is considered to be sufficient in a residential scenario and hard-limits the source routing packet overhead. A data structure containing the full network topology is maintained at a controller. Slaves may act as routers, but have typically static routes configured towards a controller. An open-source implementation called OpenZWave is avail-

[8] Adopted protocols are defined by other standard bodies such as the IETF and incorporated into the Bluetooth stack.

able on GitHub [OpenZWave, 2016]. A protocol and interoperability certification program called Z-Wave Plus is provided by the Z-Wave Alliance.

AllJoyn [AllSeen, 2016]
AllJoyn aims for Smart Home and Automotive use cases. It is an open-source framework by the AllSeen Alliance (lead by Qualcomm). It uses a D-Bus based approach in order to offer a transport agnostic service on different link layers like IEEE 802.11, Ethernet, or PLC. It targets less constrained devices running OSs like Linux, Android, Windows, or OS X. The AllJoyn terminology distinguishes between *Routers* and *Apps*, which can be hosted on the same device. Communication must always go through a *Router*. Router implementations target less constrained devices running OSs like Linux, Android, Windows, or OS X, while a so-called *AllJoyn Thin Client* should be also be able to run on a low-end IoT device. However, no porting guide for *Thin Clients* is currently available. Open-source implementations for the supported OSs are online available. Certification is provided through the AllJoyn Certified program including conformance and interoperability testing.

Summary
While the presented silo approaches can address more than one scenario, they are typically bound to a particular set of protocols and technologies, which hinders flexibility and interoperability with other systems. Thus, it is difficult, if not impossible to conduct research on alternative approaches with respect to a certain layer or a certain requirement.

2.5.2 A Clean Slate Approach: Information-Centric Networking

An alternative approach to networking in general was proposed by Van Jacobson in 2009 as *Networking Named Content* [Jacobson et al., 2009]. The resulting idea of ICN focuses on names and content instead of hosts. Different variants of this paradigm are are known under different terms, including but not limited to: Network of Information (NetInf), NDN, Content-Centric Networking (CCN), and Publish/Subscribe Networking [Ahlgren et al., 2012].

The remainder of this thesis will consider NDN, when ICN approaches are discussed. NDN follows a strict request-response pattern. Hosts that want to request a certain information, a *content chunk*, sends an *Interest* for that chunk. This *Interest* is forwarded until it reaches the content producer itself or another node that caches this content chunk. The chunk is now sent back to the requesting node, the *consumer* via Reverse Path Forwarding (RPF), following the trail of transient Pending Interest Table (PIT) entries that each intermediated node maintains. These nodes may also opportunistically cache the content chunk in their Content Store (CS), in

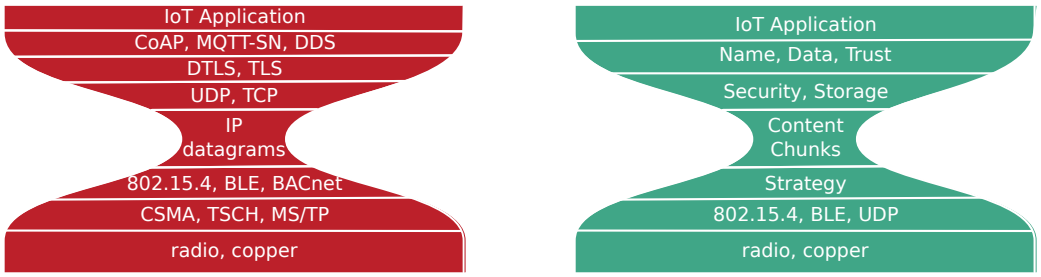


Figure 2.5: Comparison of IP IoT stack with NDN IoT stack.

order to serve potential subsequent *Interests* for this content. Content chunks represents the narrow waist of the network stack as depicted in Figure 2.5. Since each Interest retrieves at most one content chunk, the flow balance is always maintained with NDN. This flow balance is maintained at each hop and consequently no additional congestion control mechanisms are required.

Names in NDN follow a hierarchical structure. Each component of a name can describe the scope, application-specific semantics, or a specific version of the data. As a signature, the name of each content chunk is cryptographically bound to the payload at the time of production. The information about the signing key is recorded inside the data packet. This approach focusses on securing the data rather than a secure channel or session.

This paradigm recently has gained some interest for IoT use cases. It can run over multiple lower layers, e.g., either directly on the link layer, improving the payload size in LLNs, or can be encapsulated in TCP or UDP traffic. This enables transparent gateways between LLNs and the Internet. It also claims to provide inherent support for mobility and security.

Summary

ICN provides some interesting properties for the use in IoT scenarios. Chapter 6 present first analyses of large-scale ICN deployments in the IoT, the arising challenges and potentials based on experiments I have conducted. In Chapter 7 I will examine how coordinated and uncoordinated sleeping approaches can help to improve the energy efficiency of IoT applications without harming the content availability leveraging ICN's caching abilities. ICN can also be used to provide scheduling solutions to the TSCH MAC layer as I will demonstrate in Chapter 8.

2.6 Open Challenge: Energy Trade-offs

Energy efficiency is one of the most important and most challenging requirements for IoT scenarios as seen in Section 1.2. Many IoT devices are battery-powered and are expected to last years on a small battery. In the big picture: the staggering number of expected IoT devices calls for even more energy efficiency. From the hardware perspective, energy efficient MCUs and radio transceivers consume energy in mW range and provide ultra-efficient sleep modes in nW range. Additionally, energy harvesting techniques may also be applicable in some cases, but are not the focus of this thesis. The standard approach to energy efficiency for communication protocols in the IoT as presented so far consists in combining the techniques below:

- Low-power radio and MAC layers based on radio duty-cycling, aiming to reduce idle listening as much as possible.
- Less chatty network layer protocols.
- Energy-aware routing metrics.
- Offloading and proxy mechanisms.

However, saving energy often conflicts with other requirements and optimization goals. Hence, it is often inevitable to come up with a trade-off.

2.6.1 Trade-off I: Energy vs. Content Availability

In order to save the maximum amount of energy, nodes in an IoT scenario do not only have to reduce communication and switch off their radio as often as possible, but let the whole system sleep for most of the time. This demand however conflicts with the availability of content provided by the particular node. Hence, a trade-off between energy efficiency and content availability arises.

In the standard approach on networking in the IoT as described in Section 2.3, this trade-off is either addressed by offloading or proxy solutions. If permanent connectivity to the cloud or other networks not subject to strict energy constraints is available, sensors can offload their data via an uplink. Additionally, the core specification of CoAP [Shelby et al., 2014] defines a proxy that can cache resource representations for sleeping CoAP hosts. A further approach to deal with sleeping nodes in the network is the *CoRE Resource Directory* [Shelby et al., 2016]. Here, an entity called *resource directory* hosts descriptions of resources available on other, potentially sleeping, nodes in the network. Another mechanism that is proposed to allow for sleeping nodes without reducing the availability of resources is the *CoAP Publish-Subscribe Broker* [Koster et al., 2016]. In this approach a node acting as a broker enables publish-subscribe communication via store-and-forward messaging. This broker needs to be reachable by all clients at all times and requires

sufficient resources in terms of storage and bandwidth. It is responsible for hosting CoAP resources on behalf of the clients and potentially buffer messages.

In many use cases however, installing a designated gateway or proxy node is either not possible, not all sensors can reach this gateway/proxy at all times, or this gateway/proxy itself is required to also sleep a large part of the time to save energy and increase life-time. Chapter 7 focusses on studying mechanisms that dynamically distribute cached IoT content and allow IoT devices to be in sleep mode as often as possible, while maintaining acceptable levels of availability for IoT content.

2.6.2 Trade-off II: Energy vs. Latency

Several IoT use cases require bounded end-to-end latencies as seen in Section 1.2. Consequently, already the MAC layer mechanisms need to support a timely end-to-end delivery.

One way to achieve this, can be implemented by over-provisioning: **(i)** In an energy efficient, contention-based MAC protocol such as ContikiMAC [Dunkels, 2011] the duty-cycling can be increased. **(ii)** In a reservation-based MAC protocol such as TSCH this translates to a denser schedule with more reservations per neighbor. In both cases, this over-provisioning results in a higher energy consumption since the transceivers spend more time in receiving mode.

Another way to reduce the end-to-end latency with time-slotted MAC protocols like TSCH is a flow-ordered transmission schedule. If the communication path and all involved routers are known, a schedule that allows flow-ordering can be computed. In a flow-ordered schedule links are scheduled in a way so that the end-to-end latency is minimized [J.J. Garcia-Luna-Aceves and Rolando Menchaca-Mendez, 2012]. However, without a priori knowledge about communication patterns and traffic flows, it is impossible to optimize a TSCH schedule with respect to latency.

Moreover, the additional requirement of various IoT use cases for reliable end-to-end delivery poses more challenges with respect to bounded latencies. Transport layer services like confirmable messages in CoAP require retransmissions in case of packet loss. However, these retransmissions are conducted along the whole path, resulting in a higher energy consumption and a higher delay. While rather aggressive retransmission timeouts may be necessary to achieve the required end-to-end latency, they may result in superfluous transmissions, in case they are triggered too early, causing an increased energy footprint. An end-to-end acknowledgement also means that in case of a reservation-based MAC protocol, all cells on the path between source and destination need to be allocated until reception of the last fragment 6LoWPAN fragmentation/CoAP block is acknowledged.

Chapter 8 presents how information-centric reservation mechanisms can be used for TSCH networks in order to reduce the end-to-end latency while dynamically adapting to the traffic demands to improve the energy efficiency.

2.7 Summary

This chapter derives the common properties of Low-power and Lossy Networks as low-bandwidth, small packet sizes, significant packet loss, and a broadcast medium. As a result of these properties in combination with the requirements of the use cases presented in Section 1.2 the communication protocols for those networks need to be energy efficient, robust, secure, able to guarantee bounded latencies, and generate low traffic. The standard network stack for IoT use cases in these networks is based on IPv6. Consequently, this chapter describes some core mechanisms to achieve energy efficient and reliable communication in an IP-based network stack, categorized into medium access, network, routing, transport, and content aware mechanisms. For certain configurations or use cases these core mechanisms need to be complemented by auxiliary mechanisms for security, network management, clock synchronization, and link-layer transmission scheduling or an interoperability framework. Besides this IP-based IoT stack other paradigms like silo solutions or information-centric approaches exist. The last part of this chapter highlights the open challenges concerning energy efficiency and the occurring trade-offs with respect to content availability and latency.

The wide range of available mechanisms and protocols for LLNs in combination with often conflicting requirements of these networks, demands a holistic research approach. Applications have to be studied on IoT hardware in realistic deployments. As a consequence tools for testbed-based research and a software platform for protocol implementations is needed. For the latter, a requirement analysis and survey over state of the art solutions follows in the next chapter.

Chapter 3

IoT Software

Work presented in this chapter spawns from [12], which I co-authored with Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes.

The reliability of an IoT system depends essentially on the properties of the deployed OS. On the one hand, it is obvious that a failing host harms the reliability of the whole system. On the other hand, the OS has to leverage the energy saving capabilities of the hardware in order to prolong the system's lifetime. As previously mentioned, traditional OS such as Linux or BSD are not applicable on low-end IoT devices, because they cannot run on the limited resources provided on such hardware. In consequence, the IoT is plagued with lack of interoperability between many incompatible vertical silo solutions. Arguably the IoT will not fulfill its potential until a software big-bang happens, resulting in the emergence of a couple of de facto standard OSs providing consistent API & SDK across heterogeneous IoT hardware platforms.

This chapter will thus survey OSs that could become the de facto standard OS for low-end IoT devices. Specifically, in contrast to the focus of WSNs, it does not target solutions providing the smallest possible memory footprint, for a specific use case. It can be observed that solutions providing the smallest possible memory footprint are typically limited to a specific use case, and are therefore unfit for becoming the generic OS for IoT devices. In contrast, the target will be on one-size-fits-all (or at least one-size-fits-most) solutions that provide the best *level of comfort* (see below) while satisfying medium memory requirements in the order of ~10 kB of RAM or more, and ~100 kB Flash or more; i.e. devices of *Class 1* and above, according to the IETF classification [Bormann et al., 2014].

Level of comfort in this context means interoperability with the rest of the Internet including **(i)** compatibility with IP protocols from a network point of view, and **(ii)** from a systems point of view, compatibility with standard programming tools, models, and languages used on Internet hosts. This chapter focuses on open source OS, but it will also briefly survey closed source alternatives. One reason for this focus is that several of the most widespread OS for low-end IoT devices are open source, and that they offer greater possibilities to examine their design and implementation at a thorough level, as is required for this survey. A number

of additional reasons for focussing on open source will also be mentioned later in the chapter.

The remainder of this chapter is organized as follows. It starts with an analysis of the requirements which should be fulfilled by an OS for IoT devices, followed by an overview over the main OS design choices and other non-technical factors in this context. Once this background settled, a survey over the OSs that are potentially applicable follows, with the goal of being exhaustive, but brief.

3.1 Constrained Nodes: Limited Resources

Low-end IoT devices are typically very constrained in terms of resources including energy, CPU, and memory capacity. Recently, the Internet Engineering Task Force (IETF) standardized a classification [Bormann et al., 2014] of such devices in three subcategories[1] based on memory capacity[2].

- *Class 0* devices have the smallest resources (<<10 KiB of RAM and <<100 KiB Flash); e.g., a specialized mote in a WSN.
- *Class 1* devices have medium-level resources (~10 KiB of RAM and ~100 KiB Flash), allowing richer applications and more advanced features than rudimentary motes, e.g., routing and secure communication protocols.
- *Class 2* devices have more resources, but are still very constrained compared to high-end IoT devices and traditional Internet hosts.

On Class 0 devices, extreme specialization and resource constraints typically make the use of a proper OS unsuitable. Therefore, the software running on such hardware is typically developed bare-metal, and very hardware-specific.

IoT devices of Class 1 and above, however, are typically less specialized. Software can alternatively transform such a device into an Internet router [Durvy et al., 2008], host, or server, with a standard network stack and reprogrammable/interchangeable applications running on top of this stack [Pebble, 2016]. Therefore, new business models currently emerge based (partly) on portable, hardware-independent software and applications running on IoT devices of Class 1 and above. Consequently, several major companies have recently announced new OSs designed specifically to run on

[1] Note that this classification is not to be confused with Electronic Product Code (EPC). It is based on IETF standard classification as specified in RFC 7228 [Bormann et al., 2014]. The terms *Class 0–2* are used according to this classification throughout the document.

[2] Other classifications, e.g. based on energy capacities, are possible, but available memory is most crucial for the OS design.

IoT devices, including Huawei [Huawei, 2015], ARM [ARM mbed, 2015], and Google [Anand Karwa, 2015]. Indeed, on such hardware, it is often desirable to be provided with software primitives enabling easy hardware-independent code production. More generally, there is a need for APIs beyond bare-metal programming that can cater for the wide range of IoT use cases, to facilitate large-scale software development, deployment, and maintenance. Such software primitives are typically provided by an OS. This thesis will thus focus on OSs that are appropriate for Class 1 and Class 2 devices.

3.2 Requirements for IoT Software and Middleware

This section reviews the requirements for an IoT OS. These requirements are derived from the properties and challenges presented in Section 3.1. It can be observed that these requirements are in accordance with the requirements derived in Chapter 1.

Energy Efficiency

Many IoT devices will run on batteries or other constrained energy sources. For example, smart meters and other home/building automation devices are required to work for years with a single battery charge [Min et al., 2002]. On a global level, energy efficiency is also required due to the sheer number of IoT devices that is expected to be deployed (tens of billions). IoT hardware in general—MCUs, radio transceivers, sensors—provides features to operate in an energy efficient manner. However, there is no free lunch: this yields requirements on IoT software. Indeed, unless IoT software makes use of these features (e.g., putting devices into the deepest sleep mode as often as possible), energy efficiency is not achieved. Therefore, a key requirement for OSs for the IoT is **(i)** to provide energy saving options to upper layers, and **(ii)** to make use of these functions itself as much as possible, for example by using techniques such as radio duty cycling, or by minimizing the number of periodic tasks that need to be executed. For instance, a periodic system timer that schedulers use for time slicing leads to a system that never goes to deep power-down modes, and should thus be avoided if possible.

Real-Time Capabilities

Precise timing, and timely execution are crucial in various IoT use-cases e.g., smart health applications such as body area networks (BAN) with pacemakers providing wireless monitoring and control [Milenković et al., 2006, Hughes et al., 2004], or in other scenarios including actuators and/or robots in industrial automation contexts, or a Vehicular Ad-Hoc Network (VANET). An OS that can fulfill timely

execution requirements is called a Real-Time Operating System (RTOS), and is designed to guarantee worst-case execution times and worst-case interrupt latencies. Therefore, another requirement for a generic OS for the IoT is to be an RTOS, which typically implies that kernel functions have to operate with a deterministic runtime. Tasks have to meet certain deadlines in order to work correctly. An OS that is designed to fulfill these requirements is called RTOS.

Real-time systems are typically divided into hard and soft real-time systems. In a hard real-time, a task missing a deadline leads to a system failure and cannot be tolerated. Soft real-time systems will not fail if a task does not hold the deadline, but the system performance will decrease for every missed deadline.

In order to fulfill hard real-time requirements, the design of a RTOS has to guarantee worst-case execution times and worst-case interrupt latencies. The magnitude of the upper bounds for the response time is application-dependent and can vary significantly. Traditional RTOS as fully-preemptible kernels which means that the kernel can be interrupted almost anytime [Blackham, 2013]. However, even for these systems there are code paths that cannot be interrupted which is achieved by disabling interrupts. Hence, the longest duration for which interrupts are disabled has to be determined as well. The Japanese open standard for a real-time operating system, ITRON, is popular in this field, though it aims mainly for consumer electronics [ITRON, 2016].

Network Connectivity

The main point of having IoT devices, is that they can interconnect, and communicate with one another or with the Internet. IoT devices are thus typically equipped with one (or more) network interfaces. Communication techniques used in the IoT encompass not only a wide variety of low-power radio technologies (e.g., ieee802154, Bluetooth/BLE, DASH7, and EnOcean) but also various wired technologies (e.g., PLC, Ethernet, or several bus systems). Contrary to WSN scenarios [Dong et al., 2010, Saraswat and Yadav, 2010], it is generally expected that IoT devices seamlessly integrate with the Internet; i.e. can communicate end-to-end with other machines on the Internet [Jedermann et al., 2014]. The combination of (i) having to support multiple link layer technologies and (ii) having to communicate with other Internet hosts, led to the use of network stacks based on IP protocols directly on IoT devices [Palattella et al., 2013b]. A key requirement for a generic OS for the IoT is thus to support heterogeneous link layer technologies and a network stack based on IP protocols relevant for the IoT [Palattella et al., 2013b]. Furthermore, as indicated by the evolution of Linux over the years (which is an obvious example of future-proof design), it is also desirable that the OS can cater for multiple network stacks and for continuous network stack evolution.

Security and Safety

The unified IoT platform makes physical objects accessible to applications across organizations and domains. On one hand, some IoT systems are part of critical infrastructure or industrial systems with life safety implications [Stouffer et al., 2011]. On the other hand, since they are connected to the Internet, IoT devices are in general expected to meet high security and privacy standards. Beyond the overarching trust management challenge, IoT security challenges includes data integrity, authentication, and access control in various parts of the IoT architecture. Thus, a requirement (and challenge) for an OS for the IoT is to provide the necessary mechanisms (cryptographic libraries and security protocols) while retaining flexibility and usability. Last but not least, since software with a certain degree of complexity can never be expected to be 100 % bug-free, and security standards evolve (driven by various stake holders such as industry, government, consumers etc.) it is crucial to provide mechanisms for software updates on already-deployed IoT devices—and to use open source as much as possible [Hoepman and Jacobs, 2007]. Privacy means that both the content and the context around IoT data need to be protected. Since no piece of software with a certain degree of complexity can be expected to be 100 % bug free and thus may be vulnerable to attacks after deployment, it is furthermore crucial to provide facilities for software updates.

Small Memory Footprint

Compared to other connected machines, IoT devices are much more resource-constrained, especially in terms of memory. One of the requirements for a generic OS for the IoT is thus to fit within such memory constraints. While PCs, smartphones, tablets, or laptops provide Giga- or Terabytes of memory, IoT devices typically provide a few kilobytes of memory, i.e. a million times less. This observation holds both for volatile (RAM) and persistent (ROM) memory [Bormann et al., 2014]. Examples of popular IoT devices include for instance Arduino Due, TelosB motes, Zolertia Z1, IoT-LAB-M3 nodes, or OpenMote nodes [Arduino, 2016b, IoT-LAB, 2016, OpenMote, 2016, Zolertia, 2015]. In order to fit within memory footprint constraints, IoT application designers must be provided with a set of optimized libraries (potentially cross-layer) providing common IoT functionality, and efficient data structures.

Identifying the right trade-off between **(i)** performance, **(ii)** a convenient API, and **(iii)** a small OS memory footprint, is a non-trivial challenge. For example, in many cases the OS designer has to identify the sweet spot between RAM and ROM usage. Furthermore, balance must be found between sensible programming guidelines and coding conventions which must be observed on one hand, and the high degree of modularity and configurability which is desired to fit a

wide range of use cases on the other hand. Unfortunately, Moore's law is not expected to help: it is anticipated that IoT devices will get smaller, cheaper, and more energy efficient, instead of providing significantly more memory or CPU power [Mirani, 2014, Waldrop, 2016]. Therefore, in the foreseeable future, devices with a few kilobytes of memory, such as Class 1 devices [Bormann et al., 2014] which are the focus of this thesis, are likely to remain predominant in the IoT.

A MCU in a standard IoT device, as deployed in building automation, ranges from a few hundreds bytes (e.g. 8051 MCUs as used in many household devices) up to one Mbyte of RAM (e.g. ARM Cortex-M4 MCUs acting as a gateway) and has something between a few kBytes and some MBytes of persistent storage (usually ROM) [Martocci et al., 2010]. Hence, it can be observed that memory efficiency is a top priority for the design of any software for IoT scenarios and, thus, particular for the OS. Identifying the right trade-off between performance and a convenient API on the one side and a small memory footprint on the other side is a non-trivial challenge. Moreover, in many cases the OS designer also has to identify the sweet spot between RAM and ROM usage.

But not only the OS itself has to provide a low memory footprint, it should also provide any library or application developer with aids to implement in a memory-efficient manner. This can be partly achieved by sensible programming guidelines and coding conventions, but also by efficient data structures, cross layer components, and a high degree of modularity and configurability.

Support for Heterogeneous Hardware

While the diversity of hardware and protocols used in today's Internet is relatively small from an architectural perspective, the degree of heterogeneity explodes in the IoT. The large variety of use cases [Dohler et al., 2009, Martocci et al., 2010, Pister et al., 2009, Brandt et al., 2010, Rose, Karen and Eldridge, Scott and Chapin, Lyman, 2015] led to the development of a large variety of hardware and communication technologies. IoT devices are based on various MCU architectures and families, including 8-bit (e.g., Intel 8051/52, Atmel AVR), 16-bit (e.g. TI MSP430), 32-bit (ARM7, ARM Cortex-M, MIPS32, and even x86) architectures—64-bit architectures might also appear in the future [Evanczuk, Stephen, 2013]. As considered in Section 3.2 memory and CPU power can vary drastically, and so can the relation between these properties. On top of that, key system characteristics vary wildly: for example some IoT devices provide hundreds of kilobytes of RAM, but no persistent memory to store executable code (and thus generate the need to load both code *and* data into RAM). One such board is the still popular Redwire Econotag board, which is based on an Freescale MC13224V [Redwire Llc., 2015, Freescale, 2015]. Other IoT devices

are very limited in terms of RAM, but equipped with much more ROM, such as the STM32F100VC ARM Cortex-M3 MCU [ST Microelectronics, 2015]. Similarly, IoT devices can be equipped with a wide variety of communication technologies, as described below in Section 3.2. Note that such heterogeneity may even occur within a single deployment, whereby many different types of devices take part in various tasks to achieve an overall goal [Dietrich et al., 2010, Jedermann et al., 2014]. Thus, one of the requirements—and a key challenge—for a generic OS for the IoT is to support this heterogeneity in hardware architectures and communication technologies.

On the communication side, the situation is similar. Some systems use rather reliable and comparably fast, wired links, while others work on highly fragile wireless links that may exists only during a small fragment of a network's lifetime. In one scenario, nodes can communicate directly, while other applications impose the need for multi-hop communication or packets travelling through the Internet.

Thus, a generic OS for the IoT is required to support this heterogeneity and support a big set of architectures as well as providing a palette of different protocol stacks and communication options. The differences on the hardware and network level have to be abstracted as much as possible from the high level API. (Of course, some properties of the lower layers, e.g., available memory or packet round trip times, cannot be hidden completely from the application developer.)

3.3 Key Design Choices for IoT Software

The success and applicability of an OS for the IoT are influenced by technical as well as political or organizational factors. In this section, we will overview key technical OS design alternatives, as well as relevant non-technical considerations.

3.3.1 Technical Properties

Design choices concerning, e.g., the general OS model, the scheduling strategy, or hardware abstraction, have a major impact on the capabilities and flexibility of the system. This section will overview such choices and how they affect OS applicability for IoT use cases.

General Architecture and Modularity. The first design decision that has to be made for any OS is the choice of the kernel type. This choice has a major impact on the overall architecture of the system and its modularity. A generic architecture for an IoT OS is depicted in Figure 3.1. One can differentiate between an *exokernel* approach, a *microkernel* approach, a *monolithic* approach, or a hybrid approach. The main idea behind the exokernel approach is to put as few abstractions as pos-

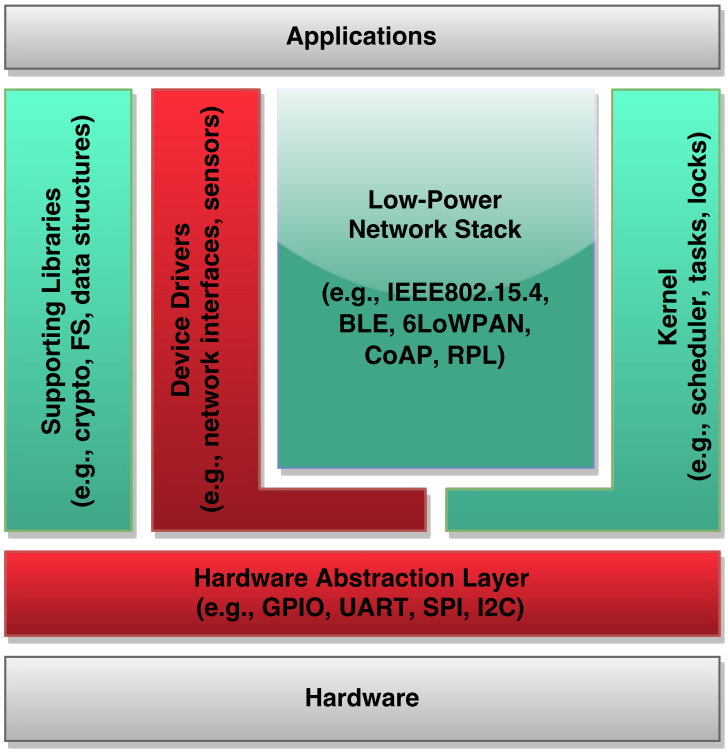


Figure 3.1: Typical components of an OS for low-end IoT devices, including a common low-power IPv6 protocol stack.

sible between the application and the hardware, and to focus on avoiding resource conflicts and checking access levels. The microkernel approach aims for more functionalities (minimalistic set of features) in the kernel, while still requiring very little memory, and providing a lot of space and flexibility for the rest of the system, as well as robustness (since a crashing device driver will not affect the stability of the whole system). However, due to the typical absence of an Memory Management Unit (MMU) on low-end IoT devices, buffer and stack overflows can still happen and have severe impact on the system. Finally, the main idea behind a monolithic approach is that all components of the system are developed together, which may lead to a simpler and overall more efficient design.

Synopsis: One has to choose between the more robust and more flexible microkernel or a less complex and more efficient monolithic kernel — or go for a hybrid approach.

Scheduling Model. Another crucial part of any OS is the scheduler, which affects other important properties such as energy efficiency, real-time capabilities, or the programming model. There are typically two types of schedulers: preemptive schedulers, and non-preemptive (or cooperative) schedulers. An OS may provide

different schedulers, that can be selected at build time. A preemptive scheduler can interrupt any (non-kernel) task at any given point to allow another task to execute for a limited time. In a cooperative model, each thread is responsible to yield itself, because no other task, and in some cases not even the kernel, is able to interrupt a task.

In many cases a preemptive scheduler requires a periodic timer tick, sometimes called a *systick*, in order to assign time slices to each task. This requirement usually prevents the IoT device to enter the deepest power-save mode, since at least one hardware timer needs to stay active. Additionally, the MCU enters full active mode at each *systick*. Time-sliced scheduling is often used for OSs with a User Interface (UI) to mimic a parallelized execution of multiple tasks. For IoT OSs this is unnecessary in most of the use cases, because they do not have a direct user and, thus, do not require a UI.

Synopsis: A preemptive scheduler assigns CPU time to each task, while the different tasks have to yield themselves in the cooperative model.

Memory Allocation. As described in section 3.2, memory is usually a very scarce resource on IoT devices. Hence, a sophisticated handling of memory is required. One important question is whether memory is allocated in a static or dynamic manner, and this choice also affects other criteria of the system design. Static memory allocation typically requires some over-provisioning and makes the system less flexible to changing requirements during run-time. Dynamic memory allocation makes the system design more complicated for two main reasons. First, functions such as `malloc()` and related functions are usually implemented in a time-wise non-deterministic fashion in the standard C libraries and, thus, will break any real-time guarantees. Hence, in order to make use of dynamic memory allocation for applications with real-time requirements, the OS has to provide special implementations for deterministic `malloc()` like TLSF [Masmano et al., 2004]. Second, dynamic memory allocation creates the need to handle out-of-memory situations and the like at runtime, which may be difficult to deal with. Additionally, heap-based malloc implementations usually induce memory fragmentation, which cause systems to run out of memory even faster.

Synopsis: Static memory allocation introduces some memory overhead due to over-provisioning and results in less flexible systems, while dynamic memory allocation leads to a more complex system and may conflict with real-time requirements.

Network Buffer Management. A central component of an IoT OS is the network stack where chunks of memory, e.g., packets, has to be shared between the layers. Two possible solutions to achieve this are copying of memory (`memcpy()`) or passing of pointers between the several layers. While the first solution is expensive from

a resource point of view, the latter generates the question who is responsible to allocate the memory. Delegating this task to the upper layers, make the application development more complex and less convenient. Leaving this task for the lower layers, such as the device driver, make the system less flexible. A possible approach to solve this conflict is the design of a central memory manager as proposed for TinyOS or RIOT [Castellani et al., 2012] [21].

Synopsis: Memory for packet handling in the network stack may be allocated by each layer or passed as a reference between the layers.

Memory Management without a MMU Another memory related problem for IoT systems result from the fact that MCUs in this domain usually do not provide a MMU or not even a memory protection unit (MPU). Thus, the protection of memory regions between multiple tasks cannot be achieved without additional overhead. However, measures can be taken to limit the effect of buffer overflows and the like by deploying stack guards, make use of minimalistic MPUs that might be present, or using a modified compiler that add safety checks to specifically annotated code, in order to prevent pointer and array errors [Cooprider et al., 2007].

Synopsis: Protections against memory corruptions without hardware support always introduce some overhead, but may be beneficial at development stage.

Programming Model. The *programming model* defines how an application developer can model the program. The typical *programming models* in the domain of IoT OSs can be divided into event-driven systems and multi-threaded systems. In an event-driven system which is, for example, widely used for WSN OSs, every task has to be triggered by an (external) event, such as an interrupt. This approach is often accompanied by a simple event loop (instead of a more complex scheduler) and a shared-stack model. A programming model based on multi-threading gives the developer the opportunity to run each task in its own thread context, and communicate between the tasks by using an Inter Process Communication (IPC) API.

Synopsis: Event-driven systems can be more memory-efficient, while multi-threading systems eases the application design.

Programming Languages. The main choice for the programming language of an OS is to decide between **(i)** a standard programming language, typically ANSI C or C++, and **(ii)** an OS-specific language or dialect. On the one hand, providing OS-specific language features allows performance- or safety-relevant enhancements that low level languages like C do not support. On the other hand, they prevent the use of well-established and mature development tools. The specification of standards for programming languages, most notably the ANSI specifications for C and C++, meant a significant boost for the evolution of software in general and for OSs in particular. Despite its age (and the rise of newer programming languages), the C programming language is still the most important and most widely used pro-

gramming language (along with Assembler) when it comes to OS programming, and to lower level parts such as scheduling or device drivers. However, more sophisticated languages with a bigger feature set may be available on top of that, at higher levels, to ease application programming.

For some of these programming languages, the OS has to provide more direct support in terms of certain timer, threading, or synchronization functionalities, for others it is more a question if certain virtual machines or scripting interpreters are available. The latter is—obviously—important for scripting languages like Perl, Python, or Ruby and languages working with byte code instead of compiling direct machine code instructions such as Java or C#.

Synopsis: Standard programming languages simplify portability and enable the use of well-known development tools. OS-specific languages and language extensions can increase the system performance and safety.

Driver Model and Hardware Abstraction Layer. IoT systems will interact with the environment in many ways, either in a passive way by sensing through all kind of sensors or actively through actuators such as motors or lighting systems. Consequently, MCUs for these systems are usually equipped with a variety of different peripheral devices, like ADCs/DACs, interfaces like SPI, I^2C, CAN bus, or serial lines, and GPIOs. Thus, a flexible and reasonably convenient driver interface is crucial for an IoT OS.

In addition to the driver model for connecting external devices, e.g., sensors, actuators, transceivers, the model may also abstract from the underlying hardware in general. A hardware abstraction layer can provide a well-defined interface to CPU, memory, and interrupt handling in order to make porting to new platforms a straightforward task.

Synopsis: A well-defined hardware abstraction layer and driver model can significantly improve the system design, but introduces a certain amount of overhead—either in terms of lines of code or in terms of runtime overhead.

Debugging Tools. As mentioned before, the choice of programming languages also predetermines the possible tools to use, including the ones for debugging. Well-established toolchains such as the one around the GNU Compiler Collection (GCC) usually include corresponding debugging tools, e.g., the GNU Debugger (GDB). However, in order to run a live debugging system, the target board has to provide an adequate interface, such as JTAG[3] or Spy-Bi-Wire. Unfortunately, not every IoT device provides such an interface, and therefore other debugging facilities are needed.

A common auxiliary tool is the use of `printf()` and the like for simple debugging over a serial interface, e.g., a USART. In some cases, even a simple LED

[3] An industry specification for on-chip instrumentation and debugging.

blinking algorithm can sometimes be found as a primitive debugging substitute. If one lacks access to the devices, as is often the case with deployed IoT networks, it is necessary to provide other means for accessing debug information. For instance, this can be achieved through periodic diagnostic messages sent over the network, or through logs written on external flash memory.

Synopsis: Using standard programming languages in general allows for using standard debugging tools, but hardware limitations may pose the need for other, simpler debugging facilities via serial output or even LED blinking.

Feature Set. An OS can be split into kernel and higher level functionalities. Typically the kernel provides a scheduler, a model for tasks, mutual exclusion (mutex) and other forms of synchronization, and timers. In case the OS supports multithreading, the API will usually also comprise functions for IPC. On higher layers, system libraries can be found, such as a shell, logging, cryptographic functions, or network stacks. Due to typically missing MMUs on IoT devices, such applications and application libraries will usually run in the same address space as kernel operations and can therefore decrease the system's stability.

In addition to network protocols, features in higher layers that are of particular interest in an OS for low-end IoT devices include over-the-air updates, dynamic loading and linking, or libraries for lightweight encryption and decryption.

Synopsis: The overall feature set of an OS may be described by the size of its API.

UI. While being of high importance for any desktop OS, most IoT OSs will not provide a graphical user interface (GUI). Reasons for that are the usual lack of a (graphical) display, the constrained device resources, and typically no direct interaction with a human user. However, many IoT OSs still provide some simpler form of UI. This can either be in form of a shell, a web interface, or other, sometimes proprietary, interface. Nevertheless, is has to be noted that these UIs are usually of less importance and typically used during development and for debugging purposes.

Synopsis: Although an UI is often not required in the productive deployment, it can become very handy during development.

Build System. When it comes to the build system, it can be observed that most IoT OSs use GNU *make* for this purpose. One reason for this is probably due to the fact that *make* is known to work well with C and is a free, mature, and well-known software that will work on almost every system in most cases. Another reason may be that more sophisticated tools often require additional effort and knowledge to set them up and are simply not required for IoT OSs, because building of these systems will usually not require a lot of resources.

Synopsis: GNU *make* is by far the most widely used build system for IoT OSs.

Testing. As for all software systems, testing plays a crucial role for the development of IoT OSs. In particular, for highly distributed development workflows, as

can often be found in bigger open source projects, deploying a continuous integration (CI) environment is inevitable [Rosenkranz et al., 2015]. This CI will usually include build and integration tests as well as unit and regression tests. The specific challenges of testing for IoT systems arise from the distributed nature of these systems, and the fact that they are deeply embedded and often very constrained. A widely used approach to deal with the hardware-related part of the testing, such as the testing of device drivers, is to use hardware emulation tools, e.g., MSPSim or Emul8 [Eriksson et al., 2007, emul8, 2016]. Network emulators and simulators such as Cooja or ns-2/ns-3, that allow for the integration of OS code, are of great help in this context [Henderson et al., 2008].

Synopsis: The distributed nature and constraints of the hardware makes thorough testing a challenging, but crucial task.

3.3.2 Non-Technical Properties

The applicability of a technically fit OS—in particular for commercial usage—is also influenced by aspects such as the license, maintainability, the workflow, or the provider of the OS. This section overviews such non-technical aspects.

(Open) Standards. A crucial characteristic for any OS is its ability to provide applications portability across hardware platforms and architectures—ideally, without any additional effort. Standardized APIs (such as POSIX, specified by IEEE and the Open Group) were also developed to simplify software porting between several OSs. However, on low-end IoT devices, implementing a standard API designed for general purpose operating systems such as Linux may be difficult because of software size constraints (and in fact, even on PCs, few OSs can claim full POSIX compliance). For seamless software porting between multiple OSs, additional support for programming language standards such as ANSI C99 or C++11 should nevertheless be provided. Finally, standards are not only important on the system level, but unavoidable on the network level. For standards at the network level, experience shows that the use of open-access specifications, such as those standardized by the IETF for instance, is preferable by default over other approaches.

Synopsis: The use of standards improves portability and interoperability.

Certification. For some use cases, in particular for critical systems in applications such as building automation, crucial properties of the system include real-time capabilities, robustness, or determinism. In these cases, certification through independent institutions becomes an inevitable requirement for the OS. A typical and widely established example for such a certification is the IEC 61508 standard, which is titled "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems". Additional certifications that are relevant for

OSs on IoT devices are the IPv6 Forum's " IPv6 Ready" logo program, and the recently started IPSO Alliance compliance and certification program.

Synopsis: Especially for the deployment in industrial and safety-critical applications, certification of the entire software running on an IoT system might be mandatory.

Documentation. Complete and easy-to-understand documentation is important for any piece of software. For an OS this requirement becomes even more important as the OS is the foundation of every other piece of software running on the system. Furthermore, need for thorough documentation is exacerbated for embedded software, as such software often has to make compromises for reasons that are difficult to grasp at first sight, due to constraints that are only partly apparent. A typical indicator for thoroughly documented code (but not necessarily the most meaningful measure) is the percentage of documentation per lines of code.

Synopsis: In order to make the best use of an OS and ease application design, a complete and comprehensible documentation is required.

Maturity of the Code. Even more difficult to measure than the quality of documentation is the maturity of software. A very rough indicator is the age of the project combined with the number of contributors and users. While certification is in many cases mainly a legal safeguard, the actual robustness and correctness of a system is much more difficult to assess.

Synopsis: In many cases thorough testing and wide deployment in commercial applications is a better indicator for the maturity of an OS than the mere age of the project or certifications.

License of the Code. In general, one can distinguish between three license categories: **(i)** non-free, **(ii)** permissive open source, and **(iii)** copyleft licenses. If an OS is released under a non-free license, the OS is either only available as binary data, or customers are charged extra fees to obtain the source code, which hampers bug fixes and improvements by third parties by limiting the number of contributors [Hoepman and Jacobs, 2007]. Permissive licenses, e.g., BSD, MIT, or Apache License, give developers and users a high degree of freedom, and are often more easily accepted by industry than copyleft licenses—although for some companies, quite the contrary is true. A possible downside of permissive licences is the potential fragmentation of the community and code base, which often leads to a situation where not all features are accessible—or at least not within one repository. By contrast, copyleft licenses such as GPL (with or without linking exception) and LGPL, are less easily accepted by some industry branches, but can lead to a much more integrative community and a common code base, as can be seen with the outstanding example of Linux.

Synopsis: Open source—particularly copyleft licenses—may not always be the first choice of industry, but offers chances for higher code quality and more secure code due to the increased numbers of contributors and reviewers.

Provider of the OS. The code of the OS may be provided in different forms and by differing entities (depending on the chosen license type). It might be either provided by the vendor that actually develops the software, or by a third party, which may also provide commercial support. In case of open source solutions, the code is often provided by the developer community itself through repositories of version control systems such as Git, Subversion, or Mercurial. The community typically provides best-effort support via online forums, open issue trackers, and mailing lists for these type of projects. This support is crucial in practice and it is thus highly recommended to prefer an open source project with a currently active community, over an open source project with no active community, or with a formerly active community. Note that sometimes, professional software consulting is offered not only for commercial OSs, but also for free open source OSs.

Synopsis: The way of distribution and degree of support for an OS is highly dependent on its license.

3.4 Candidate Operating Systems for the IoT

This section briefly reviews OSs that represent the most promising approaches towards a generic IoT OS. The goal in this section is exhaustiveness rather than in-depth analysis (which is the focus of the next section). This section distinguishes between **(i)** open source OSs, **(ii)** closed source OSs, and **(iii)** other software libraries or middleware for the IoT. If not mentioned otherwise, all OSs are written in the C programming language, while some hardware-specific parts may be implemented in assembly language. While this survey focuses mostly on (i) because an in-depth analysis of technical details is only possible for open systems, it will also briefly discuss the most relevant representatives from (ii). The first six OSs are described more in detail, because they are currently the predominant OSs in the IoT domain, mbed OS is expected to become another widely used OS in this scope, but is currently only announced. uClinux is listed as the best matching variant of the popular Linux kernel, but still fails to fulfill the IoT requirements. The remaining open source OSs are listed for the sake of completeness, but are comparably rarely deployed, and thus only their most notable properties are described. The systems in (iii) are sometimes mentioned as potential contenders, but are no real OSs.

3.4.1 Open Source Operating Systems

This section lists the predominant open source OSs targeting for IoT devices.

Contiki [Dunkels et al., 2004, Contiki, 2016] Contiki was originally developed as an OS for WSNs running on very memory-constrained 8-bit MCUs, but now also runs on 16-bit MCUs and modern IoT devices based on the ARM 32-bit MCUs. It is based on an event-driven, cooperative scheduling approach, with support for lightweight pseudo-threading. While being written in the C programming language, some parts of the OS make use of macro-based abstractions (e.g., Protothreads [Dunkels et al., 2006]), and in effect require developers to consider certain restrictions as to what type of language features they can use. Contiki code is available under BSD license on Github[4] and other platforms, while a large variety of forks are developed independently (including many closed source versions of the OS). Contiki features several network stacks, including the popular uIP stack, with support for IPv6, 6LoWPAN, RPL, and CoAP; and the Rime stack, which provides a set of distributed programming abstractions. Contiki is developed since 2002, and is so far one of the most used open source OSs for constrained nodes.

FreeRTOS [Barry, 2012] FreeRTOS is a popular RTOS which has been ported to many MCUs. Its preemptive microkernel has support for multi-threading. It is now developed by Real Time Engineers Ltd. and its code is available on the project page under a modified GPL that allows commercial usage with closed source applications (only the kernel has to remain open source). Although it does not provide its own network stack, third-party network stacks can be used for Internet connectivity. FreeRTOS is developed since 2002, and is so far one of the most used open source RTOS for constrained nodes.

TinyOS [Levis et al., 2005] Together with Contiki, TinyOS is the most prominent OS for WSN applications, targeting very constrained 8-bit and 16-bit platforms and is known for its sophisticated design. TinyOS and nesC evolved language primitives and programming abstractions to prevent as many bugs as possible through software structure and enhance memory efficiency by reducing the actual linked code to a minimum. However, the rather complex design in combination with a customized programming language makes it hard to learn, and it is thus lacking a bigger developer community [Levis, 2012]. It follows an event-driven approach, where several *components* or *modules* can be virtually wired, as described by *configurations* according the requirements. It is written in a dialect of the C programming language, called *nesC*. Its source code is available online under the BSD license on Github[5]. The included BLIP network stack implements

[4] see https://github.com/contiki-os/contiki
[5] see https://github.com/tinyos/tinyos-main

the 6LoWPAN stack. TinyOS is developed since 2000, and is so far one of the most used open source OSs for constrained nodes, with Contiki.

OpenWSN [OpenWSN, 2016] OpenWSN comprises a 6TiSCH network stack, a basic scheduler, and a *Board Support Package (BSP)* i.e. a simple hardware abstraction, making it possible to run OpenWSN on a dozen IoT hardware platforms. As such, OpenWSN is more of a network stack than a full-fledged OS. OpenWSN code is available online under the BSD license on Github[6]. The main focus of OpenWSN is the 6TiSCH network stack, including an implementation of the IEEE 802.15.4e MAC amendment [Palattella et al., 2013b]. OpenWSN is developed since 2010, by a growing, world-wide open source community.

nuttX [nuttx.org, 2015] The nuttX OS aims for full POSIX and ANSI compliance and supports MCUs ranging from 8-bit up to 32-bit architectures. NuttX can be built as a microkernel as well as a monolithic version. It is highly modular and features real-time capabilities as well as a tickless scheduler. The source code is available under BSD license on Sourceforge[7]. The integrated network stack includes support for IPv4 and IPv6 with various upper layer protocols. NuttX is developed since 2007.

eCos [eCos, 2016] The *embedded configurable operating system* (eCos) supports 16, 32, and 64-bit embedded hardware. eCos code is available under a custom license based on GPL with linking exception (acknowledged by FSF). While the open source version of eCos seems rather inactive, the commercial version (eCosPro by eCosCentric) is under active development. eCos does not provide an own network stack per se, but supports third-party network stacks (lwIP and the FreeBSD network stack). The source code is available in a Mercurial repository[8]. eCos is developed since 2002, but parts of the code-base are older.

mbedOS [ARM mbed, 2015] In 2015 ARM released a first technology preview release (labeled 15.11) of their OS for low-end IoT devices, called mbedOS. The newest release from August 2016 includes an RTOS which is based in Keil's[9] *cmsis-rtos rtx*. mbedOS focuses exclusively on 32-bit ARM embedded architecture, and supports about 20 STM, 5 NXP, and 15 platforms by other vendors. Among the features of mbedOS are a (closed-source) 6LoWPAN implementation that claims to implement the *Thread 1.0* specification, several interface definitions, a port of PolarSSL, and support for Bluetooth Low Energy. mbed is developed since 2009, but had so far focused on providing a hardware abstraction layer rather than an OS.

[6] see https://github.com/openwsn-berkeley/openwsn-fw
[7] see http://git.code.sf.net/p/nuttx/git
[8] see http://hg-pub.ecoscentric.com/ecos/
[9] Keil, renowned for their development tools for embedded software, was acquired in 2005 by ARM.

name	architecture	scheduler	programming model	targeted device class[a]	supported MCU families or vendors	programming languages	license	network stacks
Contiki	monolithic	cooperative	event-driven, Protothreads	Class 0 + 1	AVR, MSP430, ARM7, ARM Cortex-M, PIC32, 6502	C[b]	BSD	uIP, RIME
FreeR-TOS	microkernel RTOS	preemptive, optional tickless	multi-threading	Class 1 + 2	AVR, MSP430, ARM, x86, 8052, Renesas[c]	C	modified GPL[d]	None
TinyOS	monolithic	cooperative	event-driven	Class 0	AVR, MSP430, px27ax	nesC	BSD	BLIP
Open-WSN	monolithic	cooperative[e]	event-driven	Class 0 – 2	MSP430, ARM Cortex-M	C	BSD	OpenWSN
mbedOS	monolithic	preemptive	multi-threading	Class 1 + 2	ARM Cortex-M and Cortex-A, ARM7	C, C++	Apache Licence 2.0	Thread, lwIP
Zephyr	microkernel	preemptive	multi-threading or Fibers	Class 1 + 2	ARM Cortex-M3 and -M4, x86, ARC	C, C++	Apache Licence 2.0	IWIP
nuttX	monolithic or microkernel	preemptive (priority-based or round robin)	multi-threading	Class 1 + 2	AVR, MSP430, ARM7, ARM9, ARM Cortex-M, MIPS32, x86, 8052, Renesas	C	BSD	native
eCos	monolithic RTOS	preemptive	multi-threading	Class 1 + 2	ARM, IA-32, Motorola, MIPS ...	C	eCos License[f]	lwIP, BSD
uClinux	monolithic	preemptive	multi-threading	>Class 2	Motorola, ARM7, ARM Cortex-M, Atari	C	GPLv2	Linux
ChibiOS/RT	microkernel	preemptive	multi-threading	Class 1 + 2	AVR, MSP430, ARM Cortex-M	C	Triple License[g]	None
nanoRK	monolithic (resource kernel)	preemptive	multi-threading	Class 0	AVR, MSP430,	C	Dual License	None

Table 3.1: Overview of potential open source OSs for the IoT

[a] According to [Bormann et al., 2014], cf. Section 3.1.
[b] with some restrictions
[c] and several other MCUs
[d] added an exception to allow commercial use
[e] only a rudimentary scheduler is provided, support for RIOT and FreeRTOS scheduler
[f] GPL with linking exception
[g] Developer version under GPL, Releases under GPL with linking exception, commercial licensing is possible

Zephyr [Zephyr, 2016] Originally developed as Rocket (see below) by Wind River (an Intel subsidiary) the Linux Foundation adopted this project in February 2016. Zephyr supports ARM Cortex-M3 and -M4, x86, and ARC MCUs and officially supports currently 13 different platforms. It provides two types of kernels, a microkernel for slightly less constrained devices, and an underlying nanokernel. Multithreading is supported along with a lightweight, non-preemptible task model called *fiber* (comparable to Contiki's Protothreads). So far Zephyr does not provide its own network stack, but *lwIP* can be linked into a Zephyr application. Zephyr's source code is available and licensed under Apache License 2.0.

Apache Mynewt [Apache, 2016] The Mynewt OS is developed by the Apache Software Foundation since 2015. It aims to support ARM Cortex-M platforms and officially supports currently 10 ARM Cortex-M0 and Cortex-M4 boards. The architecture is based on a preemptive RTOS with support for multi-threading with fixed priorities. Mynewt OS is written in C and uses *Newt* as a command line package management and build system tool. So far it supports a BLE stack, but does not offer support for an IP stack. Support for Wi-Fi and the Thread specification are planned for later releases. The source code is online available and licensed under Apache License 2.0.

L4 microkernel family [L4, 2016, Härtig and Roitzsch, 2006] L4 OSs follow a strict microkernel design and were originally created to overcome the poor performance of earlier microkernel-based OSs in the mid-1990s. Later implementations have been designed for platform independence, improved security, isolation, and robustness. A well-known representative of this family is seL4, developed in 2006 by the NICTA group with a particular focus on security, reliability, and formal verification [Klein et al., 2009]. However, most L4 microkernel based OSs do not match the constraints of Class 1 devices. An exception is the F9 microkernel that targets particular ARM Cortex-M3/M4 based devices. While many members of this family are licensed under GPL or BSD license, not all of them are open source.

uClinux [ucLinux, 2016] This is a port of the Linux 2.x kernel for CPUs without an MMU and with a much smaller memory footprint than Linux. While uClinux benefits from the rich feature set of Linux (including APIs, a full TCP/IP stack, and excellent file system support), it has the drawback of memory requirements that do not fit low-end IoT devices, such as Class 1 devices [Bormann et al., 2014], which are the focus of this survey. The source code is available on Sourceforge[10]. uClinux is developed since 1998.

Android [Open Handset Alliance, 2015], Brillo [Google, 2015], and LittleKernel [LK, 2016] The mobile OS Android, developed by Google, is a variant of Linux, targeting mostly smartphones and tablets, but has also been used in cars, watches,

[10] see http://sourceforge.net/projects/uclinux/files/

TVs, and other consumer electronics. The concept of *apps*, accessible through online stores where users can purchase and download application software, boosted the evolution of smartphones. While the core of Android is open source—as required by Linux' GPL—many of the device drivers and hardware support is proprietary closed source code. Similarly to other Linux-based systems, Android is unable to run on low-end IoT devices such as Class 1 devices.

In 2015, Google announced Brillo [Google, 2015], a slimmed-down version of Android that will be able to run on IoT devices offering a few tens of megabytes of memory. Hence, Brillo requires considerably less hardware resources than Android. Because it is still a variant of Linux, however, it cannot be used on the low-end IoT devices that are the focus of this survey, and therefore its technical details are note expounded.

A third approach by Google to target constrained devices is LittleKernel (LK). It targets mostly ARM platforms and requires about 15–20 kB of memory just for the core. The source code is available as open source under MIT license. However it does not provide its own network stack or targets explicitly IoT scenarios.

Other Open Source Operating Systems For sake of completeness, the following lists other open source OSs. However, since they are not as prominent, they are described in less detail.

- *ChibiOS/RT* [ChibiOS, 2016] is an RTOS is developed since 2007 under a modified GPL with linking exception and aims for high performance on 8, 16, and 32-bit MCUs.
- *CooCox CoOS* [CooCox, 2016] is a free and open RTOS specifically designed for ARM Cortex-M platforms which comes along with a full-fledged IDE, developed since 2009.
- *ERIKA Enterprise* [ERIKA, 2016] is an RTOS targeted for automotive embedded systems. It supports 8, 16, and 32-bit MCUs, has support for multi-core systems and is licensed under GPL v2 with linking exception.
- *MansOS* [Strazdins et al., 2010] is another WSN OS that aims for easy developing and debugging and supports currently 8-bit AVR and 16-bit MSP430 MCUs.
- *NanoQplus* [Kim et al., 2008] developed at ETRI targets WSN Class 0 devices and provides multi-threading and a memory protection mechanism.
- *nanoRK* [NanoRK, 2016] is an RTOS for WSNs with a focus on resource reservation for tasks, developed since 2005 for MSP430 platforms.
- *Nut/OS* [NutOS, 2016] emerged from an RTOS called *Liquorice* [Liquorice, 2016], Nut/OS focusses on constrained devices with wired (Ethernet) connections.

- *RTEMS* [RTEMS, 2016] is an open RTOS with focus on open standard APIs, multiprocessor support, and hard real-time guarantees.
- There are other open source OSs from the domain of WSNs, such as *SOS* [SOS, 2016], *MANTIS OS* [Bhatti et al., 2005, Mantis, 2016], *Lorien* [Lorien, 2016] or *LiteOS* [Cao et al., 2008], but they are mostly inactive and never targeted IoT scenarios.

A detailed tabular overview of the open source OS listed above is given in Table 3.1. On the other hand, 3.2 summarizes why OSs like Contiki or FreeRTOS are a good match to most of the requirements derived in section 3.2, while other approaches such as uClinux, Arduino, and Android fail to fulfill them.

3.4.2 Closed Source Operating Systems

In addition to the aforementioned open source OSs, several closed-source OSs have characteristics suitable for IoT domain. Albeit being proprietary, some vendors offer limited access to their source code for customers, registered users, or academic institutes. These OSs, however, are often originally designed for other domains, and typically lack important features such as energy-saving mechanisms or recently standardized IoT protocols. Still, some of the closed-source OSs can be adapted to run on Class 0 and Class 1 devices. Some of the more relevant examples are listed below.

ThreadX [ThreadX, 2016] ThreadX is an RTOS developed by *Express Logic, Inc.* which has recently been acquired by ARM. ThreadX is based on a microkernel RTOS (sometimes referred to as a picokernel) which supports multi-threading and uses a preemptive scheduler. The kernel provides two techniques to eliminate priority inversion[11]: **(i)** priority inheritance that elevates the priority level of a task while executing a critical section and **(ii)** preemption threshold that disable preemption of threads below a specified priority. Additional features such as a network stack, USB support, a file system, or a GUI can be purchased as separate products.

QNX [Blackberry Ltd., 2012] Originally developed by Quantum Software Systems in 1982, QNX was acquired by Research in Motion (RIM) in 2010. It was one of the first commercially successful microkernel-based RTOS and provides a UNIX-like API. QNX's powerful IPC served as inspiration for many subsequent OSs, such as RIOT. The current version, called *QNX Neutrino*, supports numerous architectures, but none of them matching the requirements of Class 1 devices.

[11] Priority inversion is a scheduling problem where a task with a higher priority is indirectly preempted by a task with a lower priority.

VxWorks [Wind River Systems, 2015] Developed initially in 1987 by Wind River (which is now owned by Intel), VxWorks is a monolithic kernel that focusses on support for ARM and Intel platforms, including the new Quark System on Chip (SoC). VxWorks supports IPv6 and other IoT features, but lacks support for a 6LoWPAN stack, and cannot fit on constrained IoT devices as defined by RFC 7228 [Bormann et al., 2014] which are the focus of this survey.

Wind River Rocket [Wind River, 2016] Another OS developed by Wind River is Rocket which targets particular IoT scenarios. So far, Rocket supports a single hardware platform: Intel's Galileo Gen 2 board which offers several megabytes of RAM and ROM. The OS is tightly bound to using Wind River's cloud platform Helix.

PikeOS [SYSGO, 2016] PikeOS is developed since 1991 by a company called SYSGO AG (now owned by Thales). PikeOS is a microkernel-based RTOS, which provides safety and security, and acts as a hypervisor for other OSs. Originally called P4, PikeOS is a descendent of the L4 microkernel family. PikeOS provides multiple APIs, can host various guest OSs, and is certified according to several relevant standards including IEC 61508 or EN 50128.

embOS [Segger, 2015] embOS is developed by *Segger Microcontroller Systems*, a company providing development and programming tools as well as software for embedded devices. embOS is an RTOS written in ANSI C, featuring a priority-based, tickless, preemptive scheduler, and targeting various constrained 8-bit, 16-bit, and 32-bit MCUs. A network stack (including ZigBee), USB support, a GUI, and a file system are available as separate add-on products.

Nucleus RTOS [Mentor Graphics, 2015] Nucleus is an RTOS developed by Mentor Graphics, an electronic design automation company, which acquired the former provider of Nucleus, Accelerated Technology, in 2002. Nucleus enables C++ programming, is POSIX-compliant, and compatible with the Micro ITRON interface. Nucleus has a rich feature set, including an IP network stack, and can be scaled down to tens of kilobyte, but it is not among the RTOS with the smallest memory footprints, however.

Sciopta [SCIOPTA Systems AG, 2015] Sciopta is an RTOS provided by SCIOPTA Systems AG, with a focus on safety-critical applications. Its microkernel (with a direct message passing IPC) and scheduler are written in assembler. The supported architectures comprise ARM7, ARM9, ARM Cortex-M, ARM Cortex-A, and PowerPC. SCIOPTA Systems also offers additional modules for, e.g., a FAT file system or an IP-based network stack.

µC/OS-II and µC/OS-III [Micrium, 2015a, Micrium, 2015b] µC/OS-II and µC/OS-III are two versions of an RTOS provided by Micrium Inc.. These RTOS are based on a microkernel with multi-threading and IPC capabilities. In compari-

son to µC/OS-II, the version released in 2009, µC/OS-III comprises some enhanced features such as unlimited number of tasks and priorities. Additional software packages such as a GUI, a file system, or a TCP/IP network stack are also provided by Micrium, and can be integrated into µC/OS-III.

µ-velOSity [Green Hills Software, 2015] µ-velOSity is a royalty-free RTOS developed by Green Hills Software (GHS). Well integrated into Green Hills' IDE (called MULTI), µ-velOSity is written in MISRA-compliant ANSI C and based on a microkernel. Similarly to other commercial IoT OSs, additional required features (e.g., a network stack) are provided separately. Note however that a 6LoWPAN stack is not available.

Windows CE [Microsoft, 2015] Windows CE is a version of the Windows OS for constrained devices, and has been developed by Microsoft since 1996. Windows CE is real-time capable and has a rich feature set. However, it requires ROM and RAM in the order of megabytes, and therefore targets devices that are less resource-constrained than low-end IoT devices, which are the focus of this survey.

LiteOS Huawei [Huawei, 2015] In 2015, Huawei announced [Huawei, 2015] that they will release LiteOS, an operating system for IoT devices. The announcement claimed Huawei's LiteOS will fit within 10 kB of memory, and will be the most lightweight IoT OS. For now the code is not available [Huawei, 2015] and it is unclear if the OS will indeed be open source, hence it appears in the present category. Furthermore the technical characteristics of this OS are unknown, and in particular, it is unclear how it relates to the open source OS called LiteOS [Cao et al., 2008] which was mentioned in the previous section.

3.4.3 Other Software

For the sake of completeness, this section summarizes a collection of other pieces of software that are sometimes mentioned as potential contenders, but in fact are not full-fledged OSs, or are not applicable on Class 1 devices.

Arduino [Arduino, 2016a] Originating from a university project, Arduino is an open source hardware and software company. Bundled with an IDE targeting people unfamiliar with programming, it enables easy prototyping. Good support for hardware features is achieved by the fact that Arduino provides both platforms and software. Arduino does not, however, provide a real scheduler, support for threading, or any higher layer functionality, thus making it suitable primarily for simpler applications.

Espruino [Espruino, 2016] Espruino provides several embedded platforms and an open source software environment. The software part is a very efficient interpreter for JavaScript that makes it feasible to run JavaScript code on constrained devices with less than 100 kB of RAM. However, similar to Arduino, the Espruino

name	category	MCU w/o MMU	< 32 kB RAM	6LoW-PAN	RTOS sched-uler	HAL	energy efficient MAC layers
Contiki	event-driven	✓	✓	✓	✘	✓	✓
FreeR-TOS	RTOS	✓	✓	✘[a]	✓	✘	✘[b]
nuttX	multi-threading	✓	✓	✘	✓	✓	✘
uClinux	multi-threading	✓	✘	✓	✘	✓	✘
Android	multi-threading	✘	✘	✘	✘	✓	✘
Arduino	other	✓	✓	✘	✘	✓[c]	✘

Table 3.2: Key features of representatives of several categories. (✓) full support, (✘) no support. The table compares the OS for support of MCUs without MMU, MCUs with less than 100 kB of RAM, a 6LoWPAN network stack, a real-time capable scheduler, a hardware abstraction layer (HAL), and energy efficient MAC layers.

[a] available from third parties
[b] available from third parties
[c] limited portability

does not aim to replace a full-featured OS, but rather to provide a scripting framework for hobbyists and makers. It does not provide basic OS functionality such as a scheduler or thread management. Due to the nature of a scripting language, it is furthermore not capable of fulfilling real-time guarantees or fit on low-end IoT devices, but rather devices such as Tessel [tessel, 2016].

node OS [nodeOS, 2016] Node OS is a toolset written entirely in Javascript. Although its name suggests it is an OS, node OS is rather a middleware than an OS itself. It does not operate directly on the hardware, but runs on top of the Linux kernel. The requirement for Linux, coupled with the overhead of Javascript, make Node OS inappropriate for low-end IoT devices such as Class 1 devices.

3.5 Categorization of Operating Systems Relevant for IoT

The following analysis will focus on open source OSs. The reasons for this are (i) security and trustworthiness through transparency of code running on IoT devices, and (ii) the anticipated need to spread development costs between multiple parties (similarly to Linux). The open source OSs surveyed in Section 3.4 can be categorized by their architectural concept into three main categories: **(i)** pure RTOS, **(ii)** event-driven OSs, and **(iii)** multi-threading OSs. Although there is some overlap between these categories, they will define the main characteristic of an OS.

3.5.1 Pure Real-Time Operating Systems

An RTOS focuses primarily on the goal of fulfilling real-time guarantees, in an industrial/commercial context. In this context, formal verification, certification, and standardization are usually of crucial importance. To allow model checking and formal verification, the programming model used in such OSs typically imposes strict constraints for developers. These restrictions often makes the OS rather inflexible and porting to other hardware platforms may become rather difficult. Operating systems for IoT devices that fall in this category include FreeRTOS, eCos, RTEMS, ThreadX, and a collection of other commercial products (generally closed source).

3.5.2 Event-driven Operating Systems

This is the most common approach for OSs initially developed to target the domain of WSNs, such as Contiki or TinyOS for instance. The key idea of this model is that all processing on the system is triggered by an (external) event, typically signaled by an interrupt. As a consequence the kernel is roughly equivalent to an infinite loop handling all occurring events within the same context. Such an event handler typically runs to completion. While this approach is efficient in terms of memory consumption and low complexity, it imposes some substantial constraints to the programmer e.g., not all programs are easily expressed as a finite state machine [Dunkels et al., 2004]. OSs that fall in this category include Contiki, TinyOS, and OpenWSN.

3.5.3 Multi-Threading Operating Systems

Multi-threading is the traditional approach for most modern OSs (e.g. Linux), whereby each thread runs in its own context and manages its own stack. With this approach, some scheduling has to perform context switching between the threads. Each process is handled in its own thread and can, in general, be interrupted at

any point. Stack memory can usually not be shared between threads. Hence, a multi-threading OS usually introduces some memory overhead due to stack over-provisioning and runtime overhead due to context switching. Operating systems that fall in this category include nuttX, eCos, or ChibiOS.

3.5.4 Conclusion

Based on the analysis on the requirements of IoT use cases and their particular need for energy efficiency and reliability derived in Section 1.2, it can be concluded that none of the OSs that existed at the time this thesis started were a good match. Pure RTOS are a good match for the reliability aspects (and are obviously best in class to fulfill timing requirements), but are often lacking flexibility and a good support for a modular network stack. Event-driven OSs often enforces the developer to deviate from standard programming paradigms, limiting the number of tools. Besides, the often are unable to fulfill certain timing criteria. In the multi-threading category, OSs were either too big (e.g. uClinux) or lacking a proper support on the networking side (e.g. ChibiOS). As a consequence, we started to develop RIOT which is presented in detail in Chapter 4.

3.6 Summary

This chapter analyzes the various requirements to be fulfilled by an OS for low-end IoT devices, which are too resource-constrained to run traditional OSs such as Linux. It provides an overview over key aspects for such an OS, both from technical and non-technical points of view. Considering these aspects, it surveys available OSs that could qualify to become the go-to OS for IoT devices. It focusses on open source OSs because, in the context of IoT, acute privacy and security concerns are to be anticipated. Such concerns present an immense challenge that is easier to address with open source code, which offers higher potential for transparency, trustworthiness, and security. In order to benefit fully from the advantages of open source in terms of trustworthiness, it is also necessary to use open source toolchains to produce and deploy binaries on IoT devices (and to rule out dependency on untrusted third-party servers/cloud services to produce and deploy these binaries). In the long run, the collaborative nature of most open source development increases the probability that bugs are found and fits better the needs of SMEs. According to recent studies [Basiliere and Tully, 2014], such companies will be driving IoT innovation in the near future, but are more likely than bigger companies to need IoT software development and maintenance costs sharing.

This chapter covers many of the trade-offs being made by system designers regarding the requirements and constraints of current IoT applications and hardware platforms. As the IoT field is developing rapidly, however, it has to bee seen what type of architecture and capabilities an ideal OS for the IoT should have.

The work in this chapter was published in the *IEEE Internet of Things Journal* [12].

Part II

Software and Tools for Experimental Research on Energy Efficient IoT

Chapter 4

RIOT: An OS for the IoT

This chapter presents the design and implementation of general purpose software components for reliable IoT services. More specifically, it examines the RIOT operating system, its architecture, implementation details, and its network stacks. It also highlights the importance of open source and open standards in this context and how RIOT positions itself in this aspect.

It is obvious that such a huge software system as an OS together with a full-fledged network stack exceeds the work that can be achieved by a single person within the time of a doctorate. Hence, much of the work that is presented in this chapter has been conducted by many different people. However, I was involved in the conceptual design of all fundamental building blocks (except the kernel itself) and steered the development process from the beginning together with a few other persons.

4.1 A General Purpose Operating System for Reliable IoT

Work presented in this section spawns from [17], which I co-authored with Emmanuel Baccelli, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt, and from [9] which I co-authored with Emmanuel Baccelli, Hauke Petersen, Matthias Wählisch, and Thomas C. Schmidt.

In 2008, a WSN research project on tracking and monitoring vital parameters of fire fighters during emergency operations inside buildings created the need for an OS for constrained devices able to fulfill real-time requirements. The project's requirement analysis came to the conclusion that the event-driven approaches of traditional WSN OSs (compare Section 3.5.2) could not meet the requirements in terms of reactivity and flexibility [Will et al., 2009]. Consequently, a new kernel called *FireKernel* was developed. Initially intended to replace only Contiki's scheduler, most of the other parts of the OS were rewritten from scratch, too. In 2010, I realized that this microkernel could become the basis of an OS fulfilling the requirements of the emerging IoT as described in Chapter 3. Thus, I initiated the development of the 6LoWPAN stack as described in Section 2.3 accompanied with the necessary libraries on top of this kernel, naming the resulting OS µkleos. In 2013, together with colleagues from *Freie Universität Berlin* and *HAW Hamburg*,

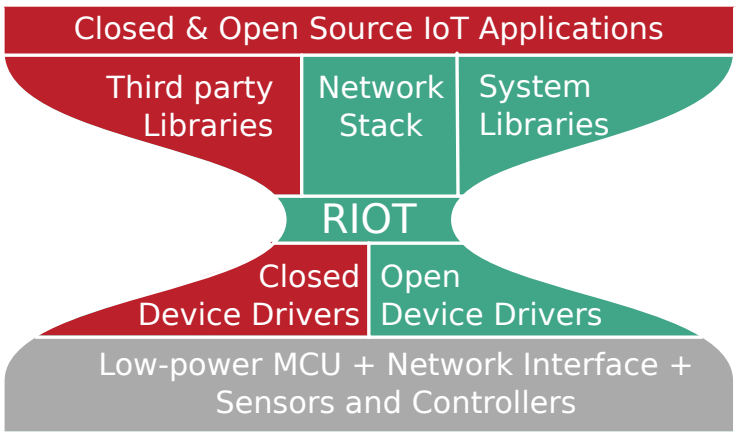


Figure 4.1: RIOT as the narrow waist for the IoT software stack.

we decided to open the development to a wider public and address other users than just academia. We published the source code, re-branded as *RIOT*, on the GitHub platform which eventually evolved into a large open-source project with over 100 contributors from all over the world.

Currently, RIOT's master branch on GitHub is maintained by several tens of developers that are in charge of reviewing and merging external contributions that are provided through *pull requests*. A lively official mailing list is also used by the community to discuss various technical and community-related matters.

At the time of writing, the community around RIOT is growing (including companies, academics, makers, and hobbyists), but smaller than that around Contiki for example [BlackDuck, 2016]. RIOT is used in various academic research institutions in the domain of IoT for both, teaching and research. Therefore, it supports hardware used on several open testbeds e.g., FIT IoT-LAB [IoT-LAB, 2016] or DES-Testbed [Günes et al., 2014]. LGPLv2.1 was chosen as a license to ensure the openness of RIOT's core components, but allow for applications and drivers under a different license.

Similar to IP in the standard IoT network stack, as presented in Chapter 2, RIOT aims to serve as the narrow waist for the IoT software stack as depicted in Figure 4.1. RIOT is based on design objectives derived in Section 3.2, including energy efficiency, small memory footprint, modularity, and uniform API access, independent of the underlying hardware.

4.1.1 Architectural Overview

As a whole RIOT was designed differently from Contiki, TinyOS, or Linux based on the following choices. A **microkernel architecture** was chosen instead of a

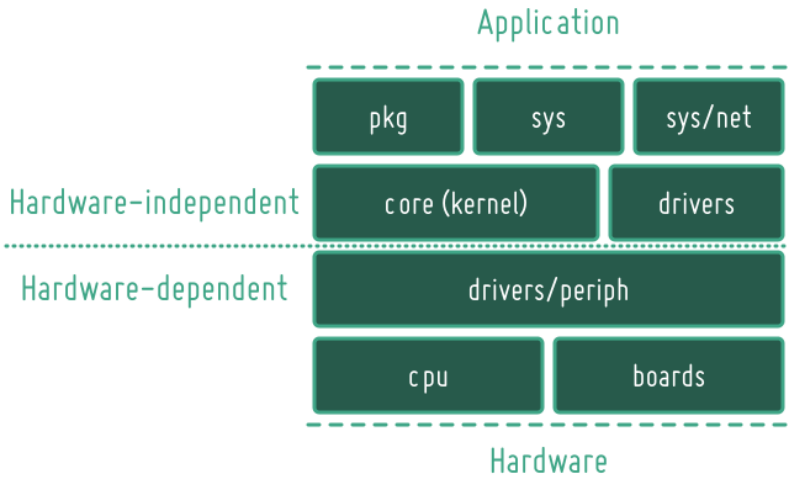


Figure 4.2: RIOT: architectural overview.

monolithic kernel. Thus RIOT can adapt more easily to the variety of IoT hardware capabilities and the variety of IoT scenarios. Minimum memory for the core system reduces to a few kBytes. RIOT's kernel is written in **ANSI C** with minor parts being implemented in assembler. However, both C and C++[1] are available as programming language for applications and application libraries. RIOT supports **multi-threading** with a memory-passing IPC between threads. Note that this programming model enables reuse of preexisting application code, without imposing exotic programming languages or predetermining particular programming models.

The RIOT **scheduler is tickless**. It does not require periodic wakeup from power-saving modes, to leverage ultra-low idle modes available on modern IoT hardware. Also, RIOT scheduler is based on **fixed priorities** and preemption, allowing for soft real-time behavior [Hennessy and Patterson, 2003]. To achieve maximum code reuse across heterogeneous IoT hardware (8-bit, 16-bit, and 32-bit), RIOT provides a well-defined **hardware abstraction layer** for peripheral interfaces as well as for networking, sensor, and actuator devices.

RIOT supports both dynamic and static **memory allocation**. However, only static methods are used within the kernel, which enables RIOT to fulfill *deterministic* requirements, by enforcing constant periods for kernel tasks (e.g., scheduler run, inter-process communication, timer operations).

RIOT introduces a **layered network stack architecture** to achieve durable network-level interoperability. The new, default stack in RIOT supports multi-

[1]Many C++ features, however, are not available on most IoT platforms due to memory constraints.

threading and a small memory footprint, for IPv6 connectivity. External stacks are integrated as well, demonstrating the power of the network APIs in RIOT.

Leveraging the fact that RIOT is written in ANSI C, well-known, established **debugging** tools can be used, such as GDB, Valgrind etc. RIOT also provides a way to run instances of the OS as processes on Linux or Mac OS, which allows both easy debugging of embedded code, and virtual network emulation. Furthermore, Cooja can also be used to simulate platforms supported by this simulator.

RIOT provides a set of unittests and applications for smoke and regression **testing**. CI testing is performed on *Murdock*, a simple CI system developed for the particular needs of RIOT [Schleiser, 2016a, Schleiser, 2016b]. Additionally, a distributed test framework was designed, in order to conduct the tests on all supported platforms [Rosenkranz et al., 2015]. Tests can also be carried out on a number of open testbeds supported by RIOT e.g., FIT IoT-LAB or DES-Testbed.

RIOT favors third-party libraries and supports a **package system** similar to BSD ports [FreeBSD, 2016], which further improves system-level interoperability in IoT compared to prior art. Finally, **open source community processes** have been implemented from the first day to favor code coherence and long-term trust at a social level, and in order to avoid related pitfalls [Levis, 2012].

4.1.2 Modularity, Configurability, Extendability

In the IoT there is no one-fits-all solution. Hence, it is inevitable to design a system in a modular and flexible manner. Many different configuration options exist and even if we just look at the network stack and consider only IP-based systems we can find a plethora of possible options. Given the constrained memory resources of a typical IoT MCU, it becomes obvious that a custom-tailored solution where the exact selection of protocols can be configured is mandatory.

From the research point of view it is also crucial to provide an extensible system. For instance, a researcher in the field of computer networks should be able to replace a certain module of the IP-stack (e.g., a RPL Objective Function (OF)[2]) without touching any other piece of the code. In that sense, extensibility is even more important than performance and efficiency. However, constrained resources do not allow to waste memory or computational power, thus, a fine balance needs to be maintained.

4.1.3 Low-Power Operation

It is the responsibility of the OS to leverage the hardware's power saving capabilities (compare Section 3.2). The kernel must decide when to switch to which Power

[2]The objective function defines how RPL nodes select and optimize routes within a RPL Instance

Mode (PM) and make use of the power saving capabilities of peripheral devices such as the transceiver, sensors, or actuators.

In order to make the best use of these energy saving capabilities, a multi-threading OS like RIOT needs to reduce the number of context switches as much as possible. The less a thread is interrupted, the faster it can complete its task and yield. As soon as all threads have completed their tasks, the scheduler can switch back to the *idle* thread. If the idle thread is scheduled, i.e. all other threads have either nothing to do or are blocked (e.g. waiting for an external event), the kernel can decide which PM to choose. This choice is based on particular dependencies of peripheral devices. The kernel has to keep track if, e.g., a timer is scheduled or any peripheral requires a running clock and cannot switch to sleeping modes that would stop the clock.

A particular challenge is given by the fact that different degrees (with varying dependencies) of sleep modes are available not only for different MCU architectures, but even among the same architecture for different vendors. Hence, the OS needs to create a reasonable mapping of internal PMs to the ones provided by the hardware.

Another challenge is the handling of PMs during debugging and experiments. Typically, a developer or experimenter make extensive use of serial input for shell commands. Serial input usually creates an interrupt. However, a MCU that is currently in sleeping mode needs a certain time to wake up. As a consequence the first character entered over the serial line is likely to get lost. Hence, the developer either needs to deactivate the power-saving modes of the OS or implement a particular mechanisms that prefixes serial input with a certain preamble and let the MCU active for a certain time.

4.2 Implementation Details

4.2.1 Microkernel Design

The core architecture of RIOT can be described as a micro-kernel in that it provides minimal functionalities: context switching, scheduling, IPC, and synchronization primitives (mutex etc.). Device drivers, network stacks, and user applications can then be run in separate thread contexts, communicating using IPC functionality provided by the kernel. This approach allows to build the complete system in a modular manner: only modules that are actually required by the application are included, thereby minimizing the memory consumption. However, modularity is kept at a coarse level (e.g. `USEMODULE += gnrc_udp` to add UDP), to avoid unman-

	Hardware Specific				
Configuration	**Platform**	**Drivers**	**Kernel**	**Net**	$\sum$
ROM					
minimal	1,754	0	854	0	2,816
WSN default	4,684	6,183	2,233	4,105	37,002
gnrc_minimal	2,732	4,106	2,140	12,298	27,524
gnrc	3,675	4,138	2,700	30,985	74,752
RAM					
minimal	656	0	2,022	0	2,880
WSN default	681	0	2,022	2,066	6,344
gnrc_minimal	676	0	2,022	2,990	7,016
gnrc	676	0	2,022	15,815	20,828

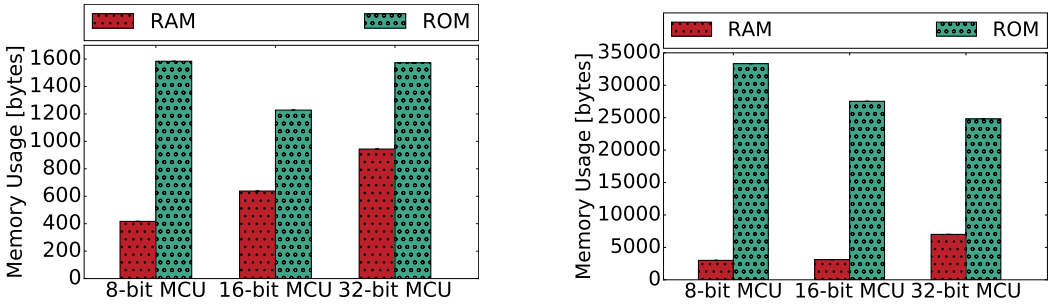
Table 4.1: Code sizes [Bytes] for different configurations and selected system components in RIOT. $\sum$ represents size of the complete RIOT image for iotlab-m3.

ageable structural complexity in the long run due to overly fine-grained software components [Levis, 2012].

Table 4.1 shows the binary code size of RIOT for different configurations on the IoT-LAB-M3 platform. Four applications are depicted: **(i)** *minimal* is most minimal version of a RIOT application including only the kernel and MCU support, **(ii)** *WSN default* is a typical WSN application including simple networking support and sensor drivers, **(iii)** *gnrc_minimal* includes a full 6LoWPAN stack (without application layer) in a minimal configuration, and **(iv)** *gnrc* is a full-fledged 6LoWPAN application as it can be used on a router in a medium-sized LLN. The *minimal* application strikingly requires 2.8 kB of RAM and 2.8 kB of ROM. However, deployment of IoT scenarios with RIOT benefit from the several hardware abstraction steps as 95 % the code is reusable and independent of the concrete hardware.

The memory requirements (RAM and ROM) for different MCU families are depicted in Figure 4.3. While architectures with smaller word size require less RAM—mostly because of the reduced stack sizes—, they may require more ROM because many operations on 32-bit variables are more expensive on 8-bit and 16-bit platforms and their toolchains apply less optimizations than the one for ARM.

No Hard Thread Separation Traditionally, on systems with MMU, OSs are divided into a kernel running in privileged mode (kernel space) where it has direct access to the underlying hardware and memory and applications running in non-privileged mode that can communicate with each other or the underlying hardware through means provided by the kernel. The constrained devices RIOT targets usu-



(a) minimal example

(b) gnrc_minimal example

Figure 4.3: Memory requirements for example applications on different platforms.

ally do not offer an MMU[3] for cost and power efficiency reasons. Thus, on most low-end IoT hardware, there is generally no separation of memory between processes, and all code on the MCU has full access to the hardware (and therefore, a kernel is not privileged with respect to other components).

Standard Multi-Threading Using RIOT's microkernel API, each component (a driver for a radio transceiver and/or some application-specific logic) can run in a separate thread context, and each thread is assigned a priority level. The amount of threads is only limited by the available memory and stack size for each thread. Multi-threading was built-in to provide several advantages: **(i)** clean logical separation between multiple tasks, **(ii)** simpler prioritization between tasks by putting them in separate threads, **(iii)** simpler distribution of tasks across multiple MCUs communicating via IPC, and **(iv)** importing code becomes easier. However, multi-threaded architectures have inherent drawbacks: on one hand per-thread management overhead and stack space, and on the other hand the overhead of IPC compared to function calls. A combination of coding conventions and RIOT microkernel design characteristics are used to mitigate these drawbacks:

Minimized Thread Management Overhead The per-thread management overhead is just 8 byte on 16-bit platforms with messaging disabled, plus 32 byte necessary for saving a thread's registers.

Minimized Stack Usage in the Code In order to keep stack usage down, code is by convention written so as to use minimum stack space, and static memory allocation is preferred.

[3]Only a small fraction of low-end MCUs offer a basic memory protection, e.g. a Memory Protection Unit (MPU)

Light-weight Inter-process Messaging The IPC was simplified to become negligible overhead with respect to actual context switch. For example, on an ARM Cortex-M0 at 48 MHz, a throughput of 100k messages/sec is achieved with 500 cycles per message. Note that MCUs on low-end IoT devices do not have advanced features (e.g., memory cache(s), out-of-order execution) which usually improve performance. However, these features make context switches more expensive. Because of that the performance impact of more-but-smaller threads is predictably down to the number of cycles needed for context saving/restoring including scheduler overhead.

Multi-threading is Optional Even though the thread management overhead is minimal in RIOT, a carefully designed single-threaded application will be most of the times faster and more memory efficient, because no IPC and no stack overprovisioning is required. Thus, for scenarios where extremely low memory usage is premium, single-threaded programming is possible on top of RIOT's microkernel API, with negligible overhead compared to not using a scheduler.

Based on this architecture, RIOT improves system-level interoperability in IoT compared to prior art, which focused on supporting event-driven programming only. In practice RIOT's microkernel itself is very small in terms of code and memory usage on supported 8-bit, 16-bit, and 32-bit IoT hardware. As seen, a minimal system based on RIOT needs 2.8 kB ROM and 2.8 kB RAM on 32bit Cortex-M platforms, including 2 kB of stack space (RAM) and about 1.5 kB of driver code (ROM).

Other functionalities necessary for a specific IoT application are added at compile-time, included as modules. A prominent example of a multi-threaded construct in RIOT is the GNRC network stack (see Section 4.3). Both GNRC's interface to applications and GNRC's internal inter-component communication are implemented using the kernel's IPC mechanism, a packet triggering 3–5 context switches (depending on GNRC configuration) while traversing up/down the stack. For the intended target applications (low-power networks) with data rates typically less than 1 Mbit/s and generally small payload (e.g. 128 byte), a typical MCU such as a Cortex-M can easily handle more than ten context switches per packet, thus using a multi-threaded design is not a bottleneck.

Thus, even a microkernel with a fully multi-threaded network stack is more than feasible on all but class 0 devices.

Real-Time and Energy Efficiency RIOT's scheduler is based on fixed priorities and preemption with $O(1)$ operations. In effect, a context switch from interrupt to a different thread will not exceed a (low) upper bound, since context saving, finding the thread to be run, and context restoring are all deterministic operations. A

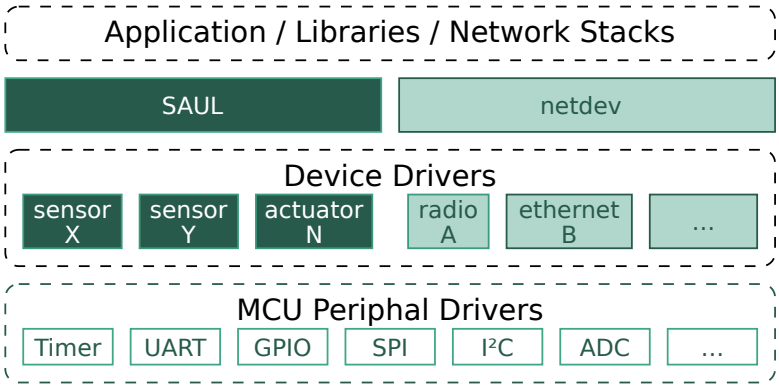


Figure 4.4: Hardware abstraction layers in RIOT.

simple scheduling policy is used: the highest priority active thread is always run, only interrupted by ISR. Combined with multi-threading, RIOT thus provides a clean way to prioritize tasks and preempt handling of low priority tasks (e.g. IP network stack interaction) in order to deal with high-priority events (e.g. engine control). The scheduling policy simplifies real-time scheduling in that, if an event requires action by a high priority thread, lower priority threads are pre-empted and the high-priority thread is run until the event has been handled. These characteristics allow for deterministic, real-time system behavior – assuming task priorities are configured coherently. In particular, there are no context switches to mimic tasks being executed in parallel, so as to minimize processor time. RIOT's tickless scheduler does not depend on artificial "ticks" (CPU time slices). Thus, the system does not require to periodically wake up unless something is actually happening, i.e. an interrupt triggered by connected hardware (e.g., from the transceiver when a packet arrived, timers firing, buttons being pressed etc.). Else, if no thread is in running state and no interrupt is pending, the system switches to the (lowest priority by default) idle thread, which can be configured to dynamically decide the deepest possible low-power mode to switch to while waiting, based on hardware capabilities.

4.2.2 Hardware Abstraction Layer

RIOT provides hardware abstraction for all building blocks of an IoT device. This includes **(i)** the MCU core, **(ii)** on-board MCU peripherals, and **(iii)** external peripheral devices. Any code that wants to interact with the hardware depends directly on one of these groups, see Fig. 4.4.

Drivers for MCU Core This API provides functionalities to the RIOT kernel, which are purely dependent on the core components of the microcontroller (ALU, handling of interrupt service routines, power management). All functionalities are implemented in C to improve readability and maintainability. Only the context switching requires assembly code.

The code is generalized and centralized for the supported MCU architectures. For example, all MCUs that are based on the Cortex-M[0-4] share the same code base. Consequently, extending RIOT to support any new Cortex-M microcontroller may reuse existing implementations of context switching and startup code.

Drivers for MCU Peripherals While memory handling is implicitly implemented by most modern hardware and the compiler, and the implementations of the MCU cores are shared for many MCU families and vendors, the MCU peripheral drivers are very specific to each vendor and even specific to certain vendor families and production cycles.

The functionality provided by the peripherals also differs widely in detail (e.g., timers can count up or down, different speeds, different number of compare channels, different counter width). However, in most cases, only a subset of the peripheral functionality is required to operate the IoT device. We thus introduce a peripheral driver layer for on-board peripheral components.

For each type of MCU peripheral (e.g., UART, SPI, GPIO) we identify basic functionalities and provide a corresponding driver interface. For example, the majority of UART devices support the 8N1 (8-bit, no parity, 1 stop bit) transfer mode. Omitting the implementation of other, less common configurations eases implementation of the interface on every MCU, independent of vendor and CPU architecture. It is worth noting that less common cases can always be supported by implementing specific peripheral drivers for a certain MCU model, if necessary. The interfaces for the MCU peripherals are designed to be light-weight. They do not provide buffering or high-level logic. Arduino and mbed offer a similar MCU peripheral driver interface. However, both implementations are based on C++, which conflicts with efficiency and flexibility as this enforces C++ applications on top of the interfaces. As a proof of concept, we implemented the Arduino API on top of the RIOT API [RIOT, 2016a].

Drivers for External Peripherals External devices are connected to the MCU via peripherals. Those peripherals only define electrical characteristics and bit encoding, but do not define any protocols and access paradigms on top of this.

Any driver for an external peripheral needs to interact with the MCU peripherals but also with the high-level applications on top, such as network stacks, user

code, and system libraries. To implement external device drivers completely independently of the microcontroller, RIOT requires that these drivers access MCU functionality only through the corresponding MCU peripheral driver interface. To make code on top of the device driver independent of the actual device in use, two high-level interfaces are proposed, SAUL (*Sensor Actuator Uber Layer*) for sensors and actuators and netdev for network devices (see Section 4.3.2).

Timer APIs Traditionally, OSs define a variety of timer interfaces, each targeting specific use case. For instance Contiki defines more than 5 timer APIs, including *etimer* for event timers, *stimer* for second timers, *ctimer* for callback timers, *rtimer* for 'real-time' timers. The consequence is increased complexity from a developer point of view, for whom it is not obvious when to use what. Consequently, RIOT defines only two timer interfaces:

- **periph/timer**: low-level peripheral timer interface, providing the slimmest possible abstraction on top of actual timer hardware, to be used for hard real-time applications.
- **xtimer**: a simple, unified timer API across all timer hardware supported by RIOT, which uses under the hood `periph/timer` for hardware abstraction.

The xtimer API takes in as parameters natural time values (e.g., microseconds). Conversion between natural time and hardware ticks is efficiently handled internally. In particular, xtimer handles timer intervals that are longer than underlying hardware timer overflow, transparently using 64-bit microsecond values. Furthermore, xtimer can multiplex a single hardware timer, and thus, the number of timers is actually limited only by memory capacity. In effect, the same mechanism is reused for thread sleep, callback timers, event timers, and could also be reused for long-term low-power timer. RIOT thus provides a portable timer API that almost completely hides the complexity of having multiple timers on diverse hardware platforms.

4.2.3 Runtime Configurability

In productive deployments many parameters can be hardwired and fixed at build time. For testing and academic research configurability at runtime is a must-have. For instance, running a series of experiments with different parameters can be done much easier and faster if nodes do not have to be reprogrammed between the several runs. Hence, providing a flexible and configurable shell was a main objective for RIOT. The RIOT shell parses single lines as commands. The maximum length of a single line can be configured at build time, but there is no limit on the number

```
> ifconfig
Iface  7   HWaddr: 1d:12  Channel: 26  Page: 0  NID: 0x23
           Long HWaddr: 36:32:48:33:46:d5:9d:12
           TX-Power: 0dBm  State: IDLE  max. Retrans.: 3  CSMA Retries: 4
           ACK_REQ CSMA MTU:1280  HL:64  6LO  RTR  IPHC
           Source address length: 8
           Link type: wireless
           inet6 addr: ff02::1/128  scope: local [multicast]
           inet6 addr: fe80::3432:4833:46d5:9d12/64  scope: local
           inet6 addr: ff02::1:ffd5:9d12/128  scope: local [multicast]
           inet6 addr: ff02::1a/128  scope: local [multicast]

           Statistics for Layer 2
            RX packets 10  bytes 388
            TX packets 3 (Multicast: 3)  bytes 156
            TX succeeded 3 errors 0
           Statistics for IPv6
            RX packets 10  bytes 598
            TX packets 3 (Multicast: 3)  bytes 178
            TX succeeded 3 errors 0

> ping6 fe80::3432:4833:46d8:7c2a
12 bytes from fe80::3432:4833:46d8:7c2a: id=83 seq=1 hop limit=64 time = 6.554 ms
12 bytes from fe80::3432:4833:46d8:7c2a: id=83 seq=2 hop limit=64 time = 8.473 ms
12 bytes from fe80::3432:4833:46d8:7c2a: id=83 seq=3 hop limit=64 time = 7.193 ms
--- fe80::3432:4833:46d8:7c2a ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2.0628106 s
rtt min/avg/max = 6.554/7.406/8.473 ms
```

Listing 4.1: RIOT's shell on IoT-LAB-M3 running a GNRC networking application. Executing `ifconfig` and `ping6`.

of parameters. Shell commands can defined on a system or an application level. Many RIOT modules define their own system level shell commands that are automatically pulled in if the application uses the shell and the respective module.

The shell itself is much less complex and powerful than even a simple POSIX shell like *dash*, since most of the more advanced features of a shell such as a command history, editing, aliases, batch execution, or timers can be easily outsource to host-side terminal tools like RIOT's *pyterm*.

Of similar importance for the runtime configurability as the shell are unified APIs of the individual modules. These APIs must provide getter/setter like functions to read and write configuration parameters at runtime. A good example for such an API is GNRC's netapi. Using netapi it is possible to configure the settings of different networking properties. If these properties should be configurable per interface, the corresponding shell command is `ifconfig`. The `ifconfig` command allows for configuring the physical as well as the logical interfaces. For example,

one can configure the radio frequency of a transceiver or set and get the IPv6 address. An example output from an IoT-LAB-M3 is depicted in 4.1[4].

4.2.4 Emulation support: RIOT as a Process

RIOT *native* is a hardware virtualizer and allows the compilation and execution of RIOT code as user processes in Linux, FreeBSD, and Mac OS X. This provides an emulator for typical IoT devices: a board and MCU featuring timers, UART, network interface, and basic sensor and actuator [5]. Virtual RIOT instances and applications can thus be run as daemons, while the virtual UART is accessible on the host OS via standard UNIX sockets and TCP. Note that, since preemptive scheduling is not available for threads within host OS user processes, a method for preemptive threading in host OS user space was designed [Ortmann, 2015].

The *native* emulator thus enables easier IoT software development, without the need for IoT hardware in the loop. Based on *native*, several RIOT instances can run in parallel on the host OS, which can communicate with one another or with the Internet through Ethernet emulator and tap interfaces. While full-mesh connectivity is provided by default between virtually networked RIOT instances on the same host OS, arbitrary topologies can easily be configured using DES-Virt [desvirt, 2016][6]. In conjunction with well-known tools as described below, *native* is heavily used by the community to speed-up testing, debug, and integration phases of IoT code – both for RIOT's hardware-independent code, and for application code running on top of RIOT.

4.2.5 Integration of Third-Party Libraries

RIOT enables the integration of third-party software and libraries as *packages*, following an approach similar to BSD ports [FreeBSD, 2016]. When building an application that uses such a package, a specified version of the library's code is automatically retrieved, e.g., from an upstream code repository, a web, or file server. Currently there is support for both Git- and HTTP-based sources, but since the integration is based on GNU Make other source types are possible, e.g. SVN. Generally, the upstream version of these libraries needs to be patched, in order to integrate seamlessly into RIOT. Packages are thus comprised of **(i)** a Makefile and a Make-

[4]Note, that the mismatch between IPv6 and Layer 2 statistics stems from 6LoWPAN HC.

[5]The terms *virtualization* and *emulation* are defined slightly different in various contexts. This thesis refers to RIOT *native* as a virtualization tool in a sense that RIOT applications and parts of RIOT can be run inside another OS to use certain features of RIOT, e.g. its network stack. It refers to *native* as an emulation tool in order to conduct experimentation and testing on IoT applications and protocols implemented for RIOT.

[6]For more information about DES-Virt, please refer to Section 5.3.2.

Package	Overall Diff Size	Relative Diff Size
libcoap	639 lines	6.3 %
libfixmath	34 lines	0.2 %
lwip	767 lines	1.3 %
microcoap	332 lines	66 %
micro-ecc	14 lines	0.8 %
relic	24 lines	<0.1 %

Table 4.2: # lines for patch files of selected RIOT packages compared to the overall LoC of the library

Application	ROM	RAM
RIOT 2016.04 gnrc_minimal + RPL + UDP	52,378	5,618
Contiki 3.0 udp-ipv6/udp-server	51,562	5,530
TinyOS tinyos-main: 177ee4b5 UDPEcho	40,574	6,812

Table 4.3: Code size comparison [Bytes] for a basic application providing standard Internet connectivity on a low-end IoT device, using RIOT, Contiki, and TinyOS.

file.include file that specify how to automatically integrate the package into RIOT and **(ii)** the collection of patch files that specify how to automatically modify the third-party software locally to make it work with RIOT. Most of the time, patch files are minimalistic, such that a package is reduced to a RIOT-specific Makefile and minor code adaptation to comply with RIOT's rather strict compiler settings. Table 4.2 shows some example of third-party software supported by RIOT using the package system. These include for example several third-party network stacks (e.g., lwIP, OpenWSN, CCN-Lite), and various libraries for crypto, math, and other such utilities (e.g., RELIC, micro-ecc, libfixmath). It is important to note that these libraries were not developed specifically for RIOT: some of these libraries were not even developed for IoT at all, but are now usable in IoT via their support as a RIOT package. Compared to prior art, RIOT thus technically enables larger scale federation of open-source communities, both within IoT and beyond IoT. Currently 14 packages are available in RIOT.

	OS Application	RIOT gnrc_minimal + RPL + UDP	Contiki udp-ipv6/udp-server	TinyOS UDPEcho
Group	Feature			
Platform	MSP430F2617 core	✓	✓	✓
	C library (newlib)	✓	✓	✓
Drivers	Timer	✓	✓	✓
	UART	✓	✓	-
	GPIO	✓	✓	✓
	SPI	✓	✓	✓
	CC2420 radio	✓	✓	✓
Network	IEEE802.15.4	✓	✓	✓
	6LoWPAN general	✓	✓	✓
	6LoWPAN fragmentation	✓	✓	✓
	6LoWPAN IPHC	✓	✓	✓
	6LoWPAN NHC	✘	✓	✓
	6LoWPAN ND	✓	✘	✓
	IPv6	✓	✓	✓
	ICMPv6 Echo	✓	✓	✓
	RPL storing	✓	✓	✓
	UDP	✓	✓	✓

Table 4.4: Comparison of features compiled into selected applications for RIOT, Contiki, and TinyOS ('✓' means present, '✘' means not present).

4.2.6 Memory Comparison to Contiki and TinyOS

It is essential checking that, though RIOT is based on radically different concepts, it is still in the same league as reference software platforms, i.e. Contiki and TinyOS. We compare at coarse-level with Contiki and TinyOS for a basic application providing standard Internet connectivity to a popular low-end IoT device (Zolertia Z1 [Zolertia, 2015] based on 16-bit MSP430 microcontroller) officially supported by all OSs. A comparison of the memory usage of the OSs is shown in Table 4.3. The binaries were obtained using GCC (`msp430-gcc 4.6.3 20120301`), using similar configuration of each OS to support basic IPv6 network connectivity featuring 6LoWPAN, RPL, and UDP. Table 4.4 shows the provided features, that are actually linked into the final binary for each OS[7]. Concerning ROM, RIOT and Contiki are on par, while TinyOS requires roughly 20 % less ROM, but this is mainly due to TinyOS not offering any standard output functions via a UART interface (using the C-libs `printf`). Concerning RAM, we observe that memory usage is similar in all

[7]Please note, that due to some minor differences in available features, the obtained numbers can not compared to the last byte.

OSs even though RIOT is based on a multi-threading programming model which requires per-thread pre-provisioning of stack memory in RAM, contrary to the programming model of Contiki and TinyOS. This might be surprising, but note that the five threads in RIOT that are used in this example do not add significant memory overhead, since stack over-provisioning and minimum per-thread stack memory allocation (52 byte on MSP430 platforms) are small. If many more threads were required, RIOT usage of RAM would of course rise significantly, but generally IoT applications need at most a handful threads.

Furthermore, we have measured the energy consumption of this basic application, used in a scenario where a sender unicasts 50 bytes UDP payload every second to a receiver within radio range, mimicking a typical observe scenario for an IoT sensor. For this measurement, we used the FIT IoT-LAB which provides open-access to precise power measurement tools on low-end IoT devices (IoT-LAB-M3). Both on the receiving and the sending nodes the power consumption is similar between RIOT and Contiki (we could not measure TinyOS which is not supported on these devices). Nevertheless, we can conclude that RIOT, Contiki and TinyOS indeed are in the same league in terms of basic resource requirements, fitting typical low-end IoT devices.

4.3 Design of the Network Stack(s)

Work presented in this section spawns from [21], which I co-authored with Hauke Petersen, Martine Lenders, Emmanuel Baccelli, and Matthias Wählisch.

The vast heterogeneity of the IoT pose many different and partly conflicting requirements to the network stack. It is thus mandatory to provide dedicated network stacks for particular use cases. For the OS this means that it must support multiple stacks in parallel, selectable and configurable at compile time. For each network stack its scope must be well-defined and is has to be designed in a self-contained manner to make it exchangeable.

4.3.1 Network Stack Requirements

RIOT's default network stack, GNRC, aims for a full-featured network stack that is flexible enough to work in a broad range of IoT scenarios, while still being efficient and small enough to run on constrained and battery-driven devices. Following similar design principles as the whole OS, also the network stack has to be designed in a maximum flexible and modular manner. Also, too many small modules would also lead to an unclear and fragmented API.

In the following, we break down the various aspects of this high-level objective.

- **Focus on IPv6.**
  The network stack should enable end-to-end connectivity between IoT devices and any other Internet device. As discussed in Chapter 2 the standard approach for this requirement is a 6LoWPAN stack. Its architecture should not exclude other protocols, but the IP suite should be the default configuration.
- **Full-featured.**
  The network stack should be full-featured in a sense that supported protocols should implement their specifications completely as a long-term goal. The point is to prohibit design decisions which will limit future extensions of an implementation. The rationale behind this is to allow for a generic solution, which can be tailored to fit various use cases, instead of a solution that is too specific by design.
- **Support for Multiple Network Interfaces.**
  IoT scenarios do not only include basic sensors with a MCU and a single low-power radio, but also border routers with multiple interfaces (e.g., Ethernet and IEEE 802.15.4) as well as upcoming IoT devices, which are likely to have multiple radio interfaces (e.g., IEEE 802.15.4 and NFC). Thus, the network stack must be able to handle multiple network interfaces, and we argue that, if designed carefully, the overhead of multi-interface support is negligible compared to single interface support, even on constrained devices.
- **Parallel Data Handling.**
  Most embedded network stacks achieve their small memory footprint by reducing their functionality, to the point where they are only able to handle a single network packet at a time. While this might be reasonable in some use cases, this is unrealistic in general. In particular, using IPv6 over spontaneous wireless networking, multiple services run in parallel, e.g., both routing and ND protocols are tightly coupled to data transfers between nodes. Thus, the network stack must be able to handle multiple packets and data streams in parallel.
- **Horizontal and Vertical Modularity.**
  The network stack consists of horizontal and vertical building blocks to implement functionality across or at the same layer. A modular network stack architecture was studied in [Ee et al., 2006] to avoid code duplication and allow run-time sharing of multiple network protocols. Their architecture, however, only focuses on the network layer and does not allow for horizontal exchange of building blocks. The heterogeneous application scenarios of IoT devices require that these building blocks can be arbitrarily combined to a complete net-

work stack. Very early work on the *x*-Kernel [Hutchinson and Peterson, 1988] provides a common API for protocol composition.

- **Loosely Coupled Components.**
  IoT network devices may provide their own network stack in addition to software implementations, as well as multiple network stacks may run in parallel on the operating system to dynamically adapt to the deployment environment. To load protocol functionalities at run time, the network subsystem should loosely combine different building blocks. All current stacks do not support such feature, either because of data sharing via inflexible callbacks or restricted APIs.
- **Configurability.**
  As for the rest of the OS the objective is the design of a versatile network stack that can be adapted to a variety of IoT scenarios. However, the granularity of configuration should avoid too many configuration options that have unclear meaning and effects (and thus are only usable for experts). Key configuration parameters must be well documented and accessible from a central point to achieve a user-friendly and flexible solution.
- **Extensibility via Clean Interfaces**
  Clean interfaces yield two important advantages. First, it yields testability by design. Second, modules and clean interfaces enable substitution of parts of the network stack, which can easily be tailored according to the IoT scenario. For example, it is straightforward to switch between two different implementations of a neighbor cache, one being optimized for run-time performance using a heap data structure, and another being optimized for memory efficiency using a simple circular list. However, again, the granularity of modules should remain coarse enough to avoid the pitfalls of ultra-fragmented code, which quickly becomes unmanageable, as analyzed in [Levis, 2012].
- **Low Memory Footprint.**
  Memory (both flash memory and RAM) is the biggest cost driver for MCUs, thus memory footprint should be kept on a leash. While the design of RIOT's network stack does not aim for the smallest possible memory footprint, it should still require very limited resources. For a concrete upper bound in accordance with the observations from 1.2 the goal is a maximum of 30 kB of ROM and 10 kB of RAM for a single interface configuration running 6LoWPAN, RPL, and UDP.
- **Low-power Design.**
  As discussed in Part I energy is one of the scarcest resources in the IoT. Experience shows that optimizations for low-power are harder to add on, and thus should built-in by design, from the very beginning. This has mainly two

consequences: **(i)** the design of the network stack must allow to easily vary the protocols used in different scenarios, as best suited, and **(ii)** the implementation must use efficient data-structures and algorithms allowing maximum sleep intervals for the CPU.

4.3.2 Network Stack Architecture

Network support is a central building block of RIOT. It is designed to be open to flexible layering and varying integration levels of network components. Beyond IP stacks, this is meant to include domain specific technologies, as well as future Internet approaches. The following will introduce the interface-centric architecture of the RIOT networking subsystem, show how this is cleanly filled by GNRC, the *generic* default network stack. The key design rule for the proposed network stack software architecture is a strict module-driven design. Emphasis is put especially on a clean definition of the interfaces between these software modules as this ensures interchangeability of modules (i.e. to choose from different implementations for different scenarios) and interoperability of these modules.

4.3.2.1 Modular Design

The top level of the software architecture consists of a number of high-level modules, one for each functional entity of the network stack, for example UDP, IPv6, 6LoWPAN, or RPL. The novelty of the proposed architecture is that each high-level module is executed in its own thread while each module services the same API utilizing the operating systems IPC. The unified interfaces allows for chaining multiple modules together and the concept is comparable to Unix *STREAMS*, as proposed in the 1980s [Ritchie, 1984], but GNRC uses IPC for communication between the modules.

The RIOT networking subsystem displays two interfaces to its externals (see Figure 4.5): The programming interface *sock* towards networking applications, and the device driver API *netdev* towards hardware. Internally, heterogeneous components interact via the unified interface netapi, thereby defining a recursive layering of a single concept that enables interaction between various building blocks: 6LoWPAN with MAC, IP with routing protocols, transport layers with the applications, etc. This grants enhanced flexibility for network devices that come with stacks integrated at different levels, and provides other advantages as outlined below. Although each of these modules can roughly be mapped to layers of the TCP/IP model, the architecture does not enforce this mapping. The APIs are presented more in detail in the subsequent Section 4.3.2.2.

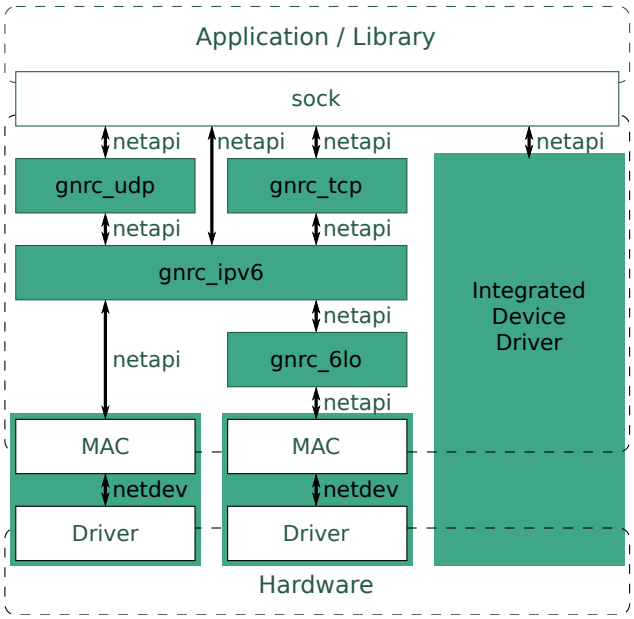


Figure 4.5: The various components of the GNRC network stack and their interactions. Each box depicts a high-level module running in it's own thread.

This design allows for a very flexible configuration of modules (even at run-time if needed) and, as important, it enables a straight-forward extension by new features or adding other layers. During design and implementation of modules this design enables further a clear separation of concerns and it enables for efficient testing of the modules. Using a unified IPC API offers further benefits when adding integrated network devices into the system that include already parts of the protocol stack, such as Texas Instrument's CC3000 which already provides a full TCP/IP stack [TI CC3000, 2015]. For a given network interface that e.g. already includes a full IP implementation, one simply needs to write a host-side device driver that can service netapi and make it known to one or more transport layer modules.

One might argue that IPC comes with a high price with respect to run-time performance and therefore energy usage. However, measurements using RIOT on state-of-the-art IoT hardware (e.g. IoT-LAB-M3) show that sending a message from one thread to another, including context save, running the scheduler and context restore, requires a number of CPU cycles that is only one order of magnitude more compared to the number of cycles needed for a direct function call. The benefits thus outweigh this overhead because **(i)** packet throughput on IoT devices is typically low, and **(ii)** there are few layers going up the stack, typically yielding IPC on less than 4 occasions.

Extensive measurements of GNRC in [Lenders, 2016] have furthermore shown that GNRC is en par in terms of memory requirements and performance. In particular, the initially stated goal of fitting into less than 30 kB of ROM and 10 kB has been reached.

4.3.2.2 Network Stack APIs

Application Interface: sock
sock is a collection of APIs to allow applications to access standard protocols via any network stack—independent of its implementation details. A POSIX socket wrapper covers *sock* in RIOT to ease portability of applications, but *sock* offers a slim way of access that is trimmed down to the specific protocol in use. It omits the overhead of the socket structure. Currently, the following APIs are defined:

- Raw IP (both IPv4 and IPv6) (`sock_ip`)
- TCP (over both IPv4 and IPv6) (`sock_tcp`)
- UDP (over both IPv4 and IPv6) (`sock_udp`)

Driver Interaction: netdev
netdev is the network device driver API used by RIOT to abstract from individual devices. It allows stacks to access network devices in a common way and offers a portable design choice of writing drivers. The driver is a set of well defined functions:

- `init()` is used to initialize a network device.
- `send()/recv()` is used to send/receive a packet over the device. It supports a handling of fragmented data.
- `get()/set()` is used to get/set an option value at the network device. Options are identified by the *netopt* data structure, an extensible type also used by netapi.
- `isr()` to inform the driver that an ISR context triggered an external event handler and requests more information on this event.

In contrary to the netapi interface this API is based on direct function calls instead of IPC. The practical reason for introducing a second interface at this stage are the tight timing constraints of MAC protocols (e.g. schemes based on TDMA). For GNRC a layer 2 specific adaptation thread runs on top of *netdev* to synchronize between function calls and IPC. Using the *netdev* API allows **(i)** for independent implementations of device drivers and MAC protocols and **(ii)** for better re-use and exchangeability of both, subsequently increasing the portability.

Inter-module Communication: netapi

The unified interface for communication between high-level modules in GNRC is called netapi. It defines typed message passing between network layers or compound modules. It was designed to be as simple and versatile as possible, so that even rather exotic network protocols can be implemented against it, and facilitate modularity to make GNRC both, easily extensible and testable. This interface is built around a small set of messages sent between the modules utilizing the operating systems IPC. The idea behind this interface is that every layer in the network stack services an identical interface. netapi supplies two asynchronous message types (`NETAPI_MSG_TYPE_SND`, `NETAPI_MSG_TYPE_RCV`) for packet data communication, and two synchronous message types that expect a reply in form of a `NETAPI_MSG_TYPE_ACK` typed message. The synchronous message types access the key-value (meta-)store of options defined by `netopt` as already in use by *netdev*:

- `NETAPI_MSG_TYPE_GET` requests an option value from a module, while the `NETAPI_MSG_TYPE_ACK` replies either the length of the option value or a negative return to report an error or that the module is not supporting the requested option.
- `NETAPI_MSG_TYPE_SET` performs analogous setter operations with corresponding returns.

Support of specific options is optional, but the reply message must indicate when an option is not supported. There is no further semantic inherent to messages of the netapi, but protocols can require certain preconditions on packets or option values handed to netapi and implement behavior that goes beyond this plain specification. This allows for versatile, yet transparent applications.Between modules, the processing chain of a packet is guided by a registry called *netreg*. An active thread can look up its succeeding thread(s) using the packet type. Modules that are interested in a certain type register with a demultiplexing context, e.g., a port number in UDP.

4.3.2.3 Packet Buffering

A key issue to solve in the design of a network stack for constrained devices is the handling of buffers for user data and protocol headers, as these are stored in RAM being one of the most limited resources (see Section 3.3.1). Typical design choices for these buffers include centralized approaches, copying data from module to module as well as mixed concepts. GNRC takes an approach with a centralized buffer and a *copy twice* concept. Outgoing data is copied once from the user application (*sock*) into a central buffer and once into a network interface's device buffer by the device driver. The same is true for receiving data in the reverse direction.

The central buffer, simply named *packet buffer*, is built to be accessible from each module through a well defined API, similar to traditional dynamic memory handling APIs such as `malloc()`. The *packet buffer* is not only able to store complete packets, but also to store unaligned fragments as single headers or data segments. This design provides major advantages in different domains: **(i)** passing around pointers to entries in the packet buffer is much more efficient than copying data around, **(ii)** it makes it simple to globally define the (maximum) amount of memory used for buffering, and **(iii)** it allows for simple exchangeability of different implementations optimized for deviating requirements.

A drawback of a single packet buffer is the missing separation of receive and send buffers, which means data flowing in one direction could use up all available buffer space, letting data flowing in the opposite direction to starve. This behavior can be countered by correct prioritization of the threads accessing the packet buffer.

The packet buffer is designed in a way that it efficiently manages this memory including mechanisms against fragmentation. For use cases where dynamic memory allocation is not an issue or even preferred, the packet buffer can simply be configure to alternatively buffer data on the heap accessed through standard library calls.

4.3.3 Third-party Network Stacks

The default network stack in RIOT as described above is GNRC. Several additional network stacks have been ported and integrated into the RIOT architecture. Some are well-known IP stacks from WSNs, some are early developments of future networking technologies. The unified APIs (see Section 4.3.2.2) allows easy integration of these stacks. If an entire stack is to be integrated, it needs only to provide a binding to *netdev* on the lower end and to *sock* on the upper end. RIOT applications can then seamlessly replace the chosen network stack. If a ported stack does additionally support *netapi* it can basically dock on any arbitrary layer. This is interesting, for e.g. ICN implementations that can use different transports and run either directly over the link layer or over UDP. In the following, we introduce four mature implementations that are included in RIOT via its package system (see Section 4.2.5).

lwIP

lightweight IP was originally developed in 2001 by Adam Dunkels [Dunkels, 2001] at SICS and is under continuous development by a worldwide community until today [Free Software Foundation, Inc., 2016]. Like GNRC, lwIP is highly modular. It uses an internal IPC API that builds upon centralized message boxes to let

both the network device and the application communicate with the central network thread. The lwIP stack supports multiple interfaces that are implemented in an object-oriented style with objects that contain both option values and methods to handle packets of different types and layers. Most protocols of the TCP/IP suite are supported including IPv4, IPv6, 6LoWPAN (added recently), TCP, and UDP.

emb6

emb6 is a fork of the Contiki uIP [Dunkels, 2003] by the University of Applied Sciences Offenburg that is working without the Contiki Protothreads [Hochschule Offenburg, 2015]. Instead it uses a sleep-and-poll scheme on event queues. emb6 is monolithic like uIP in that only a handful of feature sets can be deactivated to obtain a working configuration. It runs in a single thread and uses an event management therein to exchange packets between the layers. Only a single interface is supported, with the option to add a SLIP interface to enable IP over a UART device interface. IPv4 support was dropped from uIP, leaving full support for UDP, IPv6, and 6LoWPAN. The TCP implementation of uIP is also kept in emb6, but restricted to single segment forwarding without a sliding window.

OpenWSN

OpenWSN initially implemented at University of California, Berkeley, in 2010, is developed and maintained by a worldwide community on Atlassian under BSD license [OpenWSN, 2016]. The main focus of OpenWSN is the 6TiSCH network stack. OpenWSN is the *de-facto* reference implementation of the underlying TSCH MAC amendment [IEEE802.15.4, 2011, Palattella et al., 2013b]. The OpenWSN package in RIOT replaces the rudimentary scheduler of OpenWSN by the RIOT scheduler and adds a board support package for OpenWSN that allows it to run efficiently on top of the RIOT hardware abstraction layer.

CCN-lite

ccn-lite implements an ICN stack. It is developed mainly at the Universität Basel under ISC license since 2013. The project supports a number of packet formats, including an experimental encoding for IoT environments. With its very small core (little more than 1000 lines of C code), modular design, and support for multiple platforms, it is a very good fit for IoT requirements. Since the adapter functions for RIOT are maintained as part of CCN-lite itself, the package does not require any patches for seamless integration into RIOT. The adaptation uses the GNRC netapi and can be operated on top of any layer that provides a netapi interface. This includes transport directly over any supported link-layer protocol.

4.4 IoT Ecosystem

4.4.1 Open Standards and Interoperability

For a long time the use and deployment of many well-known standards has been considered infeasible for low-end embedded devices—both, on the system side and on the networking side. Even with the development of the first IP stacks for these systems in the early days of WSNs, it was considered rather a scientific gimmick than a realistic use case [Dunkels, 2003]. However, with the latest evolution of IoT systems and increasing relevance in commercial and governmental project, the need for standards is becoming more and more important. Interoperability and reusability are mandatory.

4.4.1.1 Standardized APIs

On the system level, there are two important standards relevant for OS development: **(i)** POSIX and **(ii)** and *ANSI C* (or *ISO C* or *Standard C*). (Actually, POSIX specifications comprise a *C POSIX library* which defines a superset of the ANSI C standard.)

The main goal for RIOT with respect to system level standards is that no additional restrictions or limitations should be raised for the developer—apart from the ones given by the hardware platform (e.g. memory constraints). This reduces the number of difficulties a developer who is new to RIOT has to overcome and allows easy integration of existing libraries (see also Section 4.2.5).

POSIX

POSIX is a set of specifications first released in 1988 to ensure compatibility between OSs. In the meantime, in addition to the core service, this family of standards comprise specifications for real-time operations, threading, as well as for shell and utilities. While several OSs even in the scope of embedded devices exist (e.g. nuttx [nuttx.org, 2015]) that strive for full POSIX compliance, it is arguable whether this is a reasonable goal for an IoT OS. In fact, not even most full-fledged desktop and server OSs can claim full POSIX compliance (although many OSs claim it). However, providing a carefully selected subset of POSIX APIs can significantly decrease development time (and costs). For RIOT we decided to analyze the POSIX dependencies of popular libraries useful in IoT use cases and implement these APIs. Following this analysis the most widely used APIs are *sockets*, *pthreads*, *mutexes*, and *semaphores*. It is noteworthy that implementing these standard APIs will generate some memory and/or computational overhead, since they were not designed for this class of devices and this type of use cases. The overhead for these

implementations varies widely and depends on the underlying OS architecture. To give two examples, we look at the implementations of *pthreads* and *sockets* for RIOT. We observe a significant overhead for *pthreads* compared to RIOT's own, less complex threading architecture. For sockets (on top of GNRC) this overhead is much smaller since most of *socket* functions can be similarly found in RIOT's connectivity API *sock*. The *pthread* implementation on RIOT consumes twice as much ROM as RIOT's own threading implementation, while the *socket* implementation consumes only 6 % of the total ROM usage in GNRC.

ANSI C

Where POSIX specifications aim to provide interoperability between different OSs, the ANSI C standards aim for compiler compatibility and portability for the C and C++ programming languages. The first standard was published by the ANSI in 1989 as *C89* (officially *X3.159-1989*), the latest version as *C11* (officially *ISO/IEC 9899:2011*) in 2011. Since not all C compiler suites support all architectures or commercial users are bound to a certain compiler, ANSI C compliance is mandatory for any software that strives for platform independence. Programming in an ANSI compliant way is mostly a matter of coding conventions and therefore does not induce any overhead. However, using features from newer standard versions or compiler dependent *directive pragmas* can lead to simpler or better readable code. Typical *directive pragmas* that are usually compatible among different compilers are `packed` or `weak`. But since they are not part of the ANSI standard, they are not safe to use for code that is intended to be portable. Hence, RIOT's coding conventions enforces compliance to the ANSI *C99* standard, implicitly prohibiting the use of any directive pragma.

4.4.1.2 Networking Standards

For an OS that targets IoT like RIOT, compliance to networking standards is at least as important as the compliance to system level standards. Important standardization bodies in the IoT scope are: IEEE, IETF, IPSO, OMA, AllJoyn, Open Interconnect Consortium (OIC), Thread, or World Wide Web Consortium (W3C). In standardization the open approach as pursued by the IETF has proven to work well for the global infrastructure of the Internet[8]. Consequently, RIOT primarily focusses on open standards, particularly since this constitutes also a good match to the open spirit of its community.

Interoperability events, like the so-called *plugtests* organized by European Telecommunications Standards Institute (ETSI) play an important role in network-

[8]In the IETF everything during the development process aims for maximum transparency and accessibility. Accordingly, all IETF documents are free and open for everyone.

ing standardization. These events are important for both, the specification developers and the implementers. While the latter can verify the interoperability of their implementations, the specification developers can check the unambiguity of their documents. For that reason RIOT has participated in four official ETSI *plugtests* between 2013 and 2015.

4.4.2 Open Source Community Aspects

4.4.2.1 Community Building

The goals that should be achieved with a widely-used open source software platform for low-end IoT devices include **(i)** enabling IoT software security and robustness in the long-term, **(ii)** enable trust, transparency, and the protection of IoT users' privacy, **(iii)** accelerate innovation by spreading IoT software development costs, and **(iv)** reduce electronic waste by preventing IoT device lock-down. The RIOT community gathers a large number of contributors [RIOT, 2016b, BlackDuck, 2016] from around the world, with various backgrounds including industry, hobbyists, and academia. Both, large companies (e.g., Cisco, Google, Intel) and SMEs (e.g., PHYTEC, Eistec, Loci Controls, Zolertia) financed some code development, while various other companies (e.g., Atmel, Nordic Semiconductors, Eclipse Foundation) sponsor or support activities around RIOT.

Building on lessons learnt with Contiki and TinyOS community building [Levis, 2012], the RIOT community organizes following an approach described below. The grass-roots RIOT community formalized a set of open processes [RIOT, 2016d], based on meritocracy, aiming at organizational durability, transparency, and code quality, partly inspired by examples from IETF and Linux communities which have proven both scalable and durable. For instance, RIOT task forces [RIOT, 2016e] gather subsets of individuals in the community on a particular technical topic, similar to IETF WGs, and, then again, RIOT maintainer status [RIOT, 2016d] giving push rights on the master branch and reviewing duties is partly inspired by Linux community organization with deputies—but without a BDFL[9]. The RIOT community used from day 1 state-of-the-art online tools (Etherpad, wiki, git, and Github) allowing open access and participation, large-scale, massively distributed revision control and source code management as well as an multifaceted documentation. Furthermore, RIOT code contribution processes mandate that the core of the software platform code remains free and licensed with a copyleft license (LGPLv2.1). This license and approach carefully chosen [RIOT, 2016c] to avoid as much as possible death-by-forking,

[9] Benevolent dictator for life

while allowing indirect business models around RIOT, similar to business around Linux [Baccelli and Schleiser, 2016].

This approach can be considered a success so far. In the first three years of existence RIOT gathered more than 100 contributors from all over the world, with a lively and helpful community (about 400 subscribers to the developer mailing list), and over 10,000 commits. In 2016 we organized the first RIOT summit gathering more than 120 people from academia, industry, and the maker scene.

RIOT's organizational characteristics are thus designed to strengthen RIOT's independence with respect to specific vendors, hardware architecture, or cloud services, and keeping the core of RIOT durably free and neutral.

4.4.2.2 Peer Reviewed Code for Enhanced Stability and Security

Open source can help to improve the quality of the software in several aspects if corresponding guidelines are taken into account [Aberdour, 2007]. Similar to the well established and renowned peer review process in scientific publication processes, we established a code review process for RIOT. Before a new piece of code is integrated into the code base, the corresponding patch is discussed and reviewed via the GitHub web interface. Each patch has to be acknowledged by at least one maintainer. For more severe changes acknowledgments from more than one maintainer are required.

This review process improves the code quality in various ways: **(i)** it ensures a consistent coding style, **(ii)** it ensures compliance with the conceptional design, and **(iii)** it reduces the probability for introducing faulty code.

The often cited "many eyes principle" can be an essential improvement on the security and privacy aspect of an IoT OS [Hoepman and Jacobs, 2007, Payne, 2002]. One the one hand, similar to the aspect of general code quality as described above, many developers and, more importantly, more reviewers increase the probability to spot a potential flaw in the code. On the other hand, the more people can review the code, the more difficult it becomes to smuggle in malicious code or back doors.

4.4.2.3 Sustainability due to an Open, Distributed Development Process

As reviewed in Section 3.4 many OSs for WSNs and Wireless Sensor and Actor Networks (WSANs) have been proposed over the last two decades. Many of these OSs follow similar design principles as RIOT, for instance MantisOS or SOS. The same can be said about various commercial OSs like ThreadX. However, none of these OSs were eligible for the research as presented in this thesis. While commercial products are often proprietary, closed-source, academic software is often tight to a particular research project or even a single thesis. Often the development is

stopped and the source code becomes unavailable after the project runtime is over or the thesis is finished.

Hence, from sustainability point of view it is important to open the development of such a software project as much as possible. Tying it to certain persons, a company, or a research project harms software reusability. Also the development of proprietary device drivers or closed source hardware support—in the worst case accompanied without freely available data sheets for the particular platform—renders a device useless if software support is terminated by the provider.

4.5 Summary and Contributions

As concluded in Chapter 3, a generic go-to software platform for IoT fulfilling its particular requirements is needed. Thus, I have initiated and (co-)designed parts of a novel OS, RIOT, addressing particularly the challenges of energy efficiency and reliability. This chapter presents the design choices—with respect to the analysis conducted in Section 3.3—we[10] have chosen to implement. It concludes that a multi-threaded micro-kernel approach is not only feasible for low-end IoT devices, but desirable to use for a large variety of IoT use cases. On top of this kernel, modularity, configurability, and interoperability were top priorities for RIOT. This chapter also shows how runtime configurability and the integration of third-party libraries increase RIOT's value to the research community. Comparing to other state of the art OSs (see Section 3.4) RIOT is en par in terms of memory requirements.

In order to provide the required networking capabilities, RIOT offers multiple solutions. On the one hand, a variety of third-party network stacks, such as lwip or CCN-lite, can be used through RIOT's packaging system. On the other hand, RIOT's own default network stack, GNRC, covers the whole range of protocols below the application layer. As a co-designer of GNRC I proposed an architecture where components are executed in separate threads and the whole stack uses only a small set of unified APIs, e.g., netapi, netdev, and sock. While this approach boldly improves the flexibility of GNRC, extensive studies have shown that it also fulfills the performance requirements.

Finally, this chapter studies the importance of an open IoT ecosystem with respect to standards and implementation. Hence, the principles of RIOT are rooted in this open ecosystem.

Contributions

As a co-founder of RIOT I have steered design decisions, the long-term strategy, and

[10] I and the other members of RIOT's core developer team.

the evolution of the open-source community. I was involved in the organization of RIOT-specific events such as Hackathons, Tutorials, and Summits. The concepts I developed or co-developed have been adopted by hundreds of peers in the research community, as well as by industrial partners.

As a core developer and maintainer of RIOT I was the responsible release manager for several releases, have contributed over 1,300 commits with more than 150,000 lines of code, supervised more than 600 pull requests.

The work in this chapter was published in the Proceedings of *IEEE Computer Communications Workshops (INFOCOM WKSHPS)* [17], the Proceedings of *IEEE International Symposium on Information Processing in Sensor Networks (IPSN)* [9], and the Proceedings of *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)* [21].

Chapter 5

Experimental Tools for Research on IoT

This chapter first analyzes which tools are required to conduct testbed-driven research of IoT scenarios. Section 5.2 discusses the challenges and limitations of testbed-driven research. This section is based on work on the *DES-Testbed* [Günes et al., 2014], which comprised over 120 nodes in 2010[1]. As a contribution, a testbed control and management framework is presented, developed for the DES-Testbed. Next, the possibilities of extending a testbed approach by virtualization and emulation are considered in Section 5.3. *DES-Virt*, a framework to conduct experiments with virtualized IoT devices in an emulated wireless network, is presented as a further contribution in this context. Finally, Section 5.4 evaluates approaches to examine a crucial property of IoT systems: the energy consumption. The contribution here is the design and implementation of a tool to evaluate the energy consumption of an IoT device in a fine-grained manner using software profiling tools. Each section contains a part about the lessons learned while using these tools.

5.1 Tools for Experiment-driven Research on IoT

Work presented in this section spawns from [4], which I co-authored with Mesut Günes, Felix Juraschek, Bastian Blywis, and Nicolai Schmittberger and from [24] which I co-authored with Bastian Blywis, Mesut Günes, and Felix Juraschek.

Conducting experiment-driven research is inevitable to study software and protocol design for IoT use cases. Experiments can be used to verify (or disprove) perceived insights gained from theoretic models or simulations. Moreover, results from these experiments can serve to generate valid input parameters for model or simulation driven research.

Many different tools are required for experiment-driven research on IoT systems. Besides the implementation of the examined approach itself, a software plat-

[1] In the meantime it became a part of the federated FIT IoT-LAB [Adjih et al., 2015].

form (e.g. an OS) to operate the IoT devices, potentially frameworks or middleware software, and tools to schedule, execute, control, and evaluate the experiment are necessary.

One of the most prominent tools for experiment-driven research in the domain of (wireless) networks is a testbed. Indeed, large-scale, persistent testbeds are a prerequisite for studying the performance of IoT software, algorithms, and protocols in an environment that resembles the reality as close as possible [Blywis et al., 2010a]. However, working on wireless testbeds can be a tedious and time-consuming task, both, for the experimenter and the testbed provider. For the experimenter, it is important that the testbed is designed in a way that it abstracts from the technical details of the testbed in order to allow focussing on the subject of the experiment instead. For the testbed provider the maintenance effort should be kept as low as possible. In Section 5.2.1 we will present a toolbox, we designed for the DES-Testbed to achieve exactly this.

Emulation is a complementary approach to address some aspects that are naturally limited in testbed experimentation. One of these aspects is scalability, another one is the controllability of the environment. The advantage of emulation over simulation here is that it resembles testbed-driven experiments as close as possible. In LLN use cases not only the hardware platform has to be emulated, but also the wireless medium itself. In Section 5.3, we present an emulation approach based on virtualized hardware and networks.

Some of the properties that need to be monitored during experimentation require additional measurement setups. Energy consumption is such a property. A combination of software mechanisms and power measuring instruments is required, in order to analyze the energy consumption. In Section 5.4, we present an approach to evaluate the energy consumption of IoT applications in a non-invasive way.

5.2 Experimentation in Large-Scale Wireless Testbeds

Testbeds are a tool of scientific research like analytical models or simulation environments. They enable studies in an environment that exhibits the same properties as a real world deployment in a production setting. In contrast to simulation environments, experiments in testbeds are subjected to uncontrollable random processes and therefore more experiment repetitions are required. The topology and quality of the links will change over time depending on environmental conditions that lead to an attenuation of the signals, e.g., humidity or groups of people that act as a black body [Shrestha et al., 2007] and due to sources of interference. The interference comes from external sources, like radios that are not part of the testbed, mi-

crowave ovens, or even sulfur lamps [van der Heijden and van der Mullen, 2002]. Additional and often more severe interference comes from within the testbed as inter and intra flow interference that is generated by data flows and management packets. Nevertheless, testbeds are especially valuable because of these issues as they can show if there are hidden or unknown problems. While protocols and algorithms can be studied in isolated and fully controllable environments, finally they have to work under totally different conditions. A study that assumes an idealized network will inevitably lead to results, that do not hold in the real world.

Simulations are run with (abstract) models for the radio propagation, mobility, and generated data flows [Wehrle et al., 2010]. The radio propagation is often modeled in a fairly simple way where the distance between two stations is the dominating (if not only) factor to determine if a communication is possible, i.e. if a link exists. Examples include the *free-space* or *two-ray ground models*. More complex models like the *shadowing model* try to consider some randomness. The general focus of such models is often on free-space propagation where obstacles and multi-path propagation are not considered. In contrast, indoor radio propagation is much more difficult to model. In this scenario, the distance will only be one of many important factors that influence the link quality. Stations are deployed in different rooms and thus each wall will attenuate the signal. The structure of the building with multiple floors, different hall and room sizes, and the position of the stations[2] relatively to their environment have to be considered. As wireless propagation in indoor environments is complex and particular models are not available or there are no commonly accepted models, testbeds are a viable tool for studies.

A testbed for IoT research needs to fulfill the following criteria:

- **sufficient size**
  The size of such a testbed is important for two reasons: **(i)** validating the scalability of an approach and **(ii)** allowing the configuration of many different topologies.
- **control and management software**
  The control and management software of a testbed is responsible for scheduling, describing, executing, monitoring, and evaluating experiments.
- **monitoring/controlling environmental parameters**
  It is not possible to provide a fully controlled environment. Hence, as a variation of the adage one could state: "Control what you can and monitor the rest".
- **open for public**
  In order to allow other researchers to reproduce results, it is mandatory that access to the testbed is open for public.

[2]More specifically: The position and orientation of the antennas are relevant.

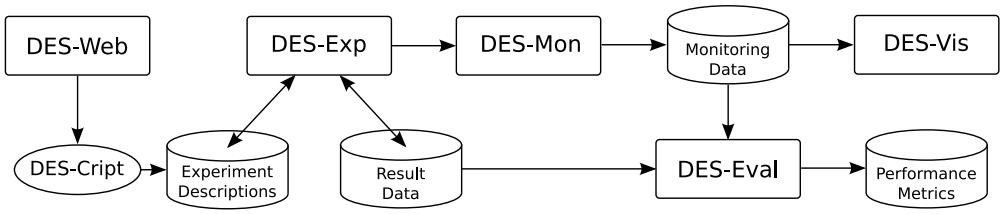


Figure 5.1: Architecture of the DES-Testbed Management System (*DES-TBMS*).

5.2.1 DES-TBMS: A Testbed Control and Management Framework

The *DES-Testbed Management System (TBMS)* was particularly designed for the *DES-Testbed* [1]. It is a framework to define, manage, coordinate, conduct, and evaluate experiments in an testbed environment and aims to simplify testbed-based research.

The DES-TBMS is similar to the *ORBIT Management Framework* (OMF) in terms of functional range [Rakotoarivelo et al., 2010]. But while the whole experiment management is set up in a distributed manner for OMF, we decided to take a centralized approach. In doing so, testbed maintenance as well as controlling the experiment is simplified. The TBMS software framework consists of six components, each dedicated to a specific task in the experimentation process. The architecture of *DES-TBMS* and the relationships between its components are depicted in Figure 5.1, a description of each component follows.

DES-Cript is a domain specific language (DSL) based on XML, which defines and describes network experiments in a holistic way. As the structure of the underlying network is abstracted, *DES-Cript* is not limited to the *DES-Testbed*. Each *DES-Cript* file contains a general information section, followed by the available network nodes assigned into groups with particular roles such as server, client, or servant. Next, actions are assigned to a group or individual nodes. As existing *DES-Cript* experiment descriptions can be edited and reused, they can be used to isolate critical parameters and can be run on different testbeds. Moreover, *DES-Cript* files provide a well-defined experiment documentation without any further effort.

DES-Exp provides an experiment manager which is responsible for the scheduling and execution of experiments. Experiments, according to the *DES-Cript* experiment description, are created and scheduled to be executed. This includes applying the settings to the *DES-Nodes* according to the chosen network configuration. *DES-Exp* also assures a defined state at the beginning of each experiment and its replications. Besides, third-party accounting and access systems like Wisebed's TARWIS [TARWIS, 2011] can use the interface of *DES-Exp* to easily integrate the *DES-Testbed* into their experimental facility.

DES-Web provides a web interface to *DES-Exp*, which allows to create, modify, and schedule experiments using *DES-Cript*.

The network monitoring tool *DES-Mon* is based on SNMP and retrieves the network state from the *DES-Nodes*. *DES-Mon* collects data from the wireless interfaces, the kernel routing table, ETX neighborhood information, and data from the sensor nodes.

DES-Vis is a 3D-visualization software based on the JavaView framework. *DES-Vis* can be used to display gathered data obtained from experiments or to show the current state of the network. For routing algorithms it can display the existing links between the network nodes or colour the links and nodes for the evaluation of channel assignment algorithms.

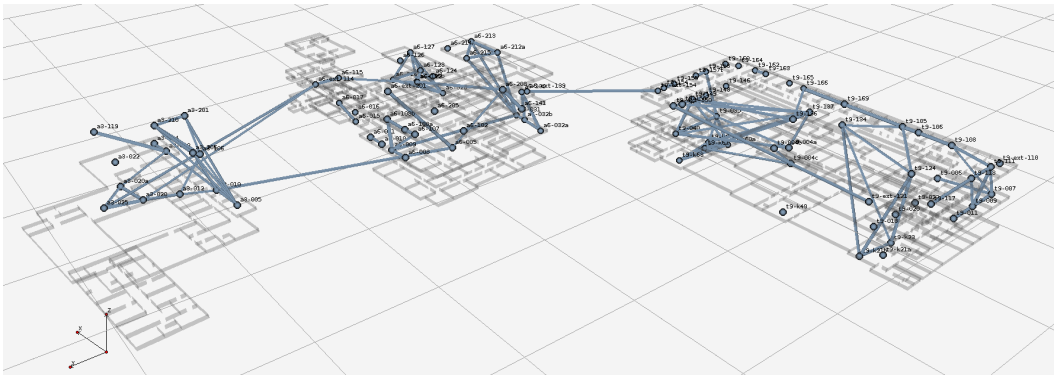
The evaluation tool *DES-Eval* enables the post-processing of the experiment results supporting work flows for an automatic evaluation process. It provides several configurable input, processing, and output modules. In addition, new modules can easily be written and integrated. Existing modules support input from log files or database records, statistical analysis using R [R, 2011], and output as plotted graphs. In this manner, experiment evaluation is made easier so that the researcher just has to combine the existing modules in a reasonable way.

5.2.2 Challenges and Limitations of Testbed-based Experimentation

One of the limitations of a testbed in comparison to simulation-based approaches is the total amount of participating nodes. Even in an advanced testbed setup, as FIT IoT-LAB [Adjih et al., 2015], the number nodes of one type per site is significantly below 1,000—and only one experimenter can use them at any given point of time. Beside the challenges of managing such a huge testbed there is also the cost factor of actual hardware.

In general, the physical topology of a testbed can only represent one certain type of deployment. Large deployments with a regular topology can mitigate this problem, as the researcher can select a subset of the available nodes to resemble the desired topology. Testbed architectures like the FIT IoT-LAB are very useful in this aspect: they offer multiple sites with the same hardware and a comparable number of nodes. This allows for conducting the same experiment in different topologies. Figure 5.2(a) depicts a testbed site where nodes are deployed in an irregular manner inside an office building, while Figure 5.2(b) shows a deployment in a regular, symmetric topology.

Filtering mechanisms like *iptables* on Linux can also be used to configure particular topologies, departing from the physical conditions. However, these forced topologies are very error-prone. It can easily happen to misconfigure the filtering mechanisms, resulting problems that are hard to trace. Moreover, these forced



(a) 3D visualization of the topology of the deployment, consisting in ≈120 nodes that interconnect via wireless communications (sub-GHz) and that are physically distributed in multiple rooms, multiple floors, and multiple buildings.

(b) Lille testbed is deployed over a 225 m^2 area, composed of a corridor separating a big room and 5 offices. Nodes are dispatched over the ceiling and wood poles, situated at the borders of the big room.

Figure 5.2: Physical topologies of different testbed deployments.

topologies will still behave differently than the topology they are trying to mimic. For example, even if a node cannot receive packets from its physical neighbor due to certain filter rules it will still suffer from interference caused by this node.

A further issue with testbeds is induced by the deployed hardware platforms. Often this hardware is not available for purchase, making it difficult for other researchers to deploy the experiments in their favoured environment. Moreover, the deployed devices are usually not upgraded periodically, which leads to testbeds using legacy hardware over time. As a consequence, many of the available LLN testbeds nowadays still use typical WSNs MSP430 platforms, such as the TelosB,

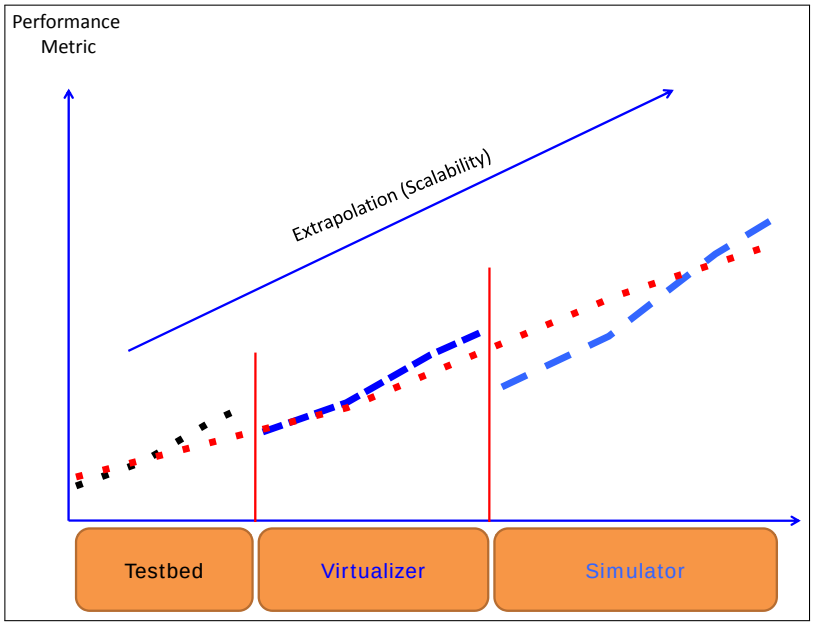


Figure 5.3: The testbed can be extended using virtualization and simulation

while the majority of IoT applications rather target ARM Cortex-M platforms. However, replacing the hardware in a large-scale testbed is not only a big cost factor, but is also a time consuming and tedious task.

Mostly the same type of hardware is deployed in these testbeds. In real IoT deployments, however, a wide heterogeneity of very different platforms exist, particular in consumer areas like home automation systems. Deployments with widely differing hardware capabilities pose many interesting challenges to IoT research, but are difficult to study with current testbed deployments. The DES-Testbed and FIT IoT-LAB address this issue by deploying different types of hardware in one deployment.

Another problem is that the variety of hardware platforms in large-scale IoT testbeds is fairly limited. As mentioned before, TelosB (and very similar variants like TMote Sky or FIT IoT-LAB's WSN430) are still used in many testbeds, while many other platforms are not available in any testbed. The FIT IoT-LAB addresses this aspect, too, by allowing users to deploy custom hardware at selected sites. These additional platforms are then accessible through the same framework as the regular FIT IoT-LAB nodes.

Virtualization is one approach to mitigate some of the difficulties mentioned above. Whereby research on networks in the range of about one thousand nodes is possible. Using virtualized nodes also allows the creation of any arbitrary topology. We will review some software and network virtualization tools in the following

Section 5.3. Simulators even go one step further and make networks in the range of ten thousand nodes feasible. On the other hand simulations require migrating the original software into the simulator while virtualized nodes can run exactly the same system.

Having all three testing facilities (testbed, virtualization, and simulation) using the same platform for experiment description, execution, and evaluation enables the comparison of results. In this way we are able to make inferences between these facilities in both directions. In terms of, for instance, scalability of an algorithm this enables new potential (see Figure 5.3).

5.2.3 Lessons Learned

Certainly, the most prominent lessons learned from experiment-driven research on IoT: it will always take (much) longer than estimated in the beginning. This might seem to be a rather trivial insight and valid for many other areas, but it is still worth to highlight here, because of the particular challenges of this approach:

- **Faulty Hardware**
  Dysfunctional or misbehaving hardware components are often a cause of trouble in experiment-driven research, but for low-end embedded devices this is particular true. Hardware is often in a rather prototypal state, revealing undocumented problems or behavior that contradicts the datasheets. Due to the low cost-factor of these boards, they are also not very robust and prone to die early.
- **Faulty Software**
  Where other researchers can rely on some mature basis components, a researcher in the IoT cannot even rely on mature hardware (see above), let alone mature software systems. Neither for the OSs nor the device drivers (even if provided by the vendor) the number of bugs is close to the amount you would expect on traditional desktop or server systems. Even for rather stable and well established systems like Linux and IEEE 802.11 problems occur where a device driver offer certain options (e.g., enabling link layer retransmissions for IEEE 802.11), but simply ignore configurations from the caller without returning an error value.
- **Fragile Environment**
  It is a well-known fact that wireless communication is prone to interferences and jamming leading to a high degree of packet loss. Beyond that, wireless testbed environments suffer from additional issues. If the nodes of a testbed are not deployed in a strictly secured environment—which is (a) not always feasible and (b) not always desirable—it has to be expected that nodes are

(re)moved, damaged, or even destroyed. And not only people moving and acting in the testbed environment, but even weather and climate have an impact: fog, rain, and snow or trees with and without leaves have a significant impact on wireless communication. All of this is very little controllable and requires huge efforts and a deep understanding to monitor. As a consequence, a researcher has to expect that a suiting topology he/she has identified, may not be available any more in the very next moment.

- **Shared Resources**
  Boldly spoken research based on theoretical models requires only pen and paper (and a trash bin) and simulations require only a (powerful) PC. Then again, in testbed-driven experiments researchers have to share the resources of the testbed among each others. This results in two challenges: **(i)** proper planning and reservation of the necessary resources is mandatory (particular with closing deadline) and **(ii)** deal with potential interferences from concurrent experiments (if the testbed architecture allows for concurrent usage).

Of course, all the usual difficulties and challenges of experiment-driven research apply, too, but won't be discussed here.

5.3 Virtualization Tools for IoT Software

Tools virtualizing both IoT hardware and networks, can extend the capabilities of testbed-driven research and help to overcome some of the limitations.

5.3.1 Virtualizing IoT Hardware and Wireless Networks

Virtual machines are an important step in between simulators and testbed experimentation. They can be seen as a particular form of *emulation*. The advantage of emulation over simulation is that the "real" software can be used, i.e. (almost) no change to the software that can be used for testbed experiments is required. The main advantages of virtualization over testbeds are: **(i)** experiments can be run in a controlled environment, **(ii)** nodes are much less constrained (e.g. almost unlimited memory), and **(iii)** snapshots of the whole system are possible.

Another advantage of virtualization compared to real hardware are the improved possibilities for debugging, testing, memory leak detection, and profiling. Low-end embedded devices often provide only very limited possibilities for debugging tools. Some platforms are only programmable over a serial bootloader and do not provide a debug interface, e.g. because the potentially available JTAG pins are not accessible. Additionally, typical MCUs only offer a limited number of hardware *breakpoints*. The use of runtime analysis tools such as memory checkers

like *valgrind* [Seward et al., 2004] or profiling tools like *gprof* [Graham et al., 1982] is also often not possible on embedded devices. Furthermore, it has proven advantageous for the analysis of networking protocols to use packet analyzers such as Wireshark [Orebaugh et al., 2006]. However, examining packet traces in real wireless deployments can be difficult, particular if not all nodes are within the same broadcast domain and multiple radio sniffers are necessary. In that case the clocks of these sniffers have to be very accurately synchronized and duplicates of the sniffed packets have to be detected and eliminated in a post-processing step. Furthermore, sniffers have to use the same antennas as the nodes participating in the experiment to guarantee an identical perception of radio signals.

However, it is important to keep in mind that the virtualized system behaves differently in some aspects due to the virtualized hardware. The main differences that can be observed are **(i)** different **system timings**, **(ii)** different **networking timings**, and **(iii)** different **memory usage**. The different **system timings** are caused by the virtualized system calls and depend on the load of the host system, i.e. also on the number of virtualized nodes. The different **networking timings** are caused by the virtualized medium. In reality wireless communication can be seen as a bus system that only allow broadcast transmissions and that all nodes can access completely asynchronously. However, in approaches as described in this thesis, a virtual medium is represented by a single entity that nodes can only access in a sequential manner. Finally, the different amount of available memory, potential different width of data types, and different implementations of standard libraries, lead to the different **memory usage**. As a result, an experimenter typically needs to assign more memory to virtualized systems than the original platform has, which in turn make it more difficult to diagnose memory leaks on the real system.

Another type of virtualized wireless connectivity is taken by some testbed federations projects such as Wisebed [Chatzigiannakis et al., 2009]. This *Virtual Distributed Testbed* uses an overlay network to provide a virtually unified testbed. Hence, any node in one of the participating testbeds can connect to any other node in one of the other testbeds through the testbeds' portal servers. This allows for bigger topologies that are assembled by federated smaller testbeds. The overlay can be created on different layers, e.g. emulating a wireless link between the portal servers.

5.3.2 DES-Virt: a Virtualization Framework for the IoT

DES-Virt [desvirt, 2016] is a Python-based virtualization framework that combines Linux' bridge capabilities with *ebtables* [ebtables, 2016] and *tc* [tc, 2016]. The main use case for *DES-Virt* is two-fold: **(i)** it should act as an emulation and extension of the DES-Testbed and **(ii)** it should be able to emulate different IoT scenarios. Con-

sequently, *DES-Virt* currently supports two types of virtual machines: Linux Kernel Virtual Machines (KVMs) [Qumranet, 2006] and RIOT *native*.[3] For Linux KVMs a server with 8 CPU cores can virtualize up to 1000 routers. From the *DES-TBMS* point of view, there is no difference between a virtualized router and a real router. For RIOT *native* typical modern desktop PC (Intel Core i7 @ 3.4 GHz with 16 Gb RAM) can handle more than 1000 instances. We leveraged this feature for experiments presented in Chapter 7 to study the scalability of our approaches.

The requirements for *DES-Virt* were as follows:

- **definition of topologies**
  Standard topologies such as *fully meshed*, *star*, and *line* topologies should be configurable as well as arbitrary topologies to mimic real-world deployments.
- **PDR-to-link assignment**
  The PDR of any link needs to be configurable. Different PDRs per direction and for unicast and broadcast transmissions must be considered.
- **data rates**
  Different data rates on a per-link basis should be supported.
- **medium access**
  The virtualizer should reflect busy medium situations and signal it to the virtual transceiver driver accordingly.
- **interference**
  Ideally a network virtualization tool would consider dependencies between links inside a network and virtualize interferences accordingly. It should be possible to configure external interferences as well.

We managed to fulfill the first three requirements, while the latter two are only implemented as a prototype so far. Using simple XML-based topology descriptions, any desired topology can be defined. As of now, it is possible to define a PDR per link, direction, and data rate. The PDR can be defined for unicast and broadcast transmission separately. A transmission delay can be configured, too. An additional tool to generate XML-files with some standard topologies is also provided.

Both, for KVM and RIOT *native*, virtual transceivers are represented as TUN/-TAP interfaces. Hence, the virtualized transceiver appears as a standard Ethernet interface to the virtualized system.

Opposed to network simulators, network virtualization needs to compute transmissions in real-time. This makes the emulation of a realistic wireless medium and model interferences so challenging. In a proposed extension of this virtualization framework a central dispatcher is introduced to enable interference and medium access emulation [Martin Nieto, 2011]. Virtual wireless interfaces are used

[3] For more information about RIOT *native* see Section 4.2.4.

to connect the virtual machines to each other or to the real testbed. Based on the `mac80211_hwsim` kernel module, they present themselves like real wireless interfaces to userspace, offering channel and rate selection, ad-hoc and infrastructure modes. Packets sent over virtual wireless interfaces are repacked into UDP packets and then sent over the management Ethernet backbone to a central wireless emulation server.

5.3.3 Lessons Learned

Running many extensive experiments in virtualized environments as provided by *DES-Virt* throughout this thesis, it showed that this approach can be very helpful to shorten development time, both for new software components as for new protocols and algorithm. It allows for an evaluation of implementations that can be applied without changes to the testbed in a controlled environment. Two small case studies can serve to depict the usefulness of this toolset.

In the first case, we consider a bug that is triggered by a race condition in the software (or protocol design), causing the affected node to fail fatally. It may happen that this race condition happens so rarely and independent from the runtime of the network, that already identifying how to reproduce the failure can result in a very time-consuming task. Attaching a debugger to every physical node in a testbed environment is impracticable. However, in a virtual environment this can be done, revealing the full back trace of the failure. Furthermore, additionally compiled in debug information may not be feasible on real hardware due to memory constraints, but may help to identify the source of the problem using the emulator.

In the second case, we consider a bug that is caused by a memory leak. While it is typically considered to be harmful to make use of dynamic memory allocation on embedded systems [Walls, 2013], it may happen that you want to re-use a library that was not particularly designed with these constraints in mind. Spotting the source for memory leaks without a *memchecker* tool can be tedious task on unconstrained systems, but may turn quickly into a nightmare, if not even proper debugging capabilities are available—which is often the case for low-end IoT devices. Now running the same software in a virtualized environment with tools like *valgrind* available can significantly shorten this process.

However, a lesson learned from these experiments in a virtualized environment was also that you should not expect the software to run flawlessly in the real testbed just because it did so in the virtualizer. The virtual hardware behaves differently and so does the virtual network.

5.4 Online, in-situ Energy Profiling

Work presented in this section spawns from [3], which I co-authored with Stephan Adler, Nicolai Schmittberger, and Mesut Günes, and from [6] which I co-authored with Stephan Adler.

Performing fine-grained analyses of the energy consumption of an IoT application in a non-intrusive manner constitutes a helpful tool in order to gain a deeper understanding of the system.

5.4.1 Evaluation of Energy Consumption

The topic of evaluation of energy consumption for WSNs is almost as old as the research area itself. To tackle this issue the traditional three concepts are used: **(i)** theoretical analysis, **(ii)** simulation, and **(iii)** using real hardware and testbeds Each of these approaches has its own advantages and disadvantages. Thus, it is necessary to combine the methods for a holistic evaluation.

Theoretical analysis allows for varying all kind of parameters in a fully controllable environment. The analysis can be done on a per-system basis as well as for huge-scale networks as a whole. Simulation still offers a high degree of controllability and enables direct examination of the impact of the implementation. While these first two methods make the results very reproducible, they have to rely on certain assumptions and input data (often taken from manufacturer's data sheets) that are hard to verify. In contrast, hardware measurements provide real data gathered from a complete system, but is prone to measurement inaccuracies and errors. Moreover, the amount of data is often hard to correlate to the system's state, particularly when whole networks are going to be evaluated.

5.4.1.1 Theoretical Analysis

The theoretical approach tries to formalize common hardware operations and create a detailed model to predict energy consumption. Early power estimation models [Simunic et al., 1999] derived the consumption of components directly from the data-sheet information and examination is done per component. Later work tried to obtain models at an instruction-level [Tan et al., 2002].

While research on WSN evolved more and more, the characteristics of low power transceivers where considered more in detail. For example, analysis for power consumption of the transceiver have been split up into formulas for the Power Amplifier (PA), Low Noise Amplifier (LNA), and RX/TX circuits [Wang et al., 2006]. This lead to the shown initial formulas 5.1 and 5.2 which

break energy consumption down to one equation for transmitting P_T and one for receiving P_R.

$$P_T(d) = P_{TB} + P_{TRF} + P_A(d) = P_{T0} + P_A(d) \tag{5.1}$$

$$P_R = P_{RB} + P_{RRF} + P_L = P_{R0} \tag{5.2}$$

Here $P_A(d)$ depicts the power consumption of the PA as a function of the transmission range, d. P_{TB} and P_{RB} denotes the power consumption of the baseband DSP[4] circuit for transmitting respectively. P_{TRF} and P_{RRF} are the power consumed in front-end circuit for transmitting and receiving. P_L describes the energy consumption of the LNA, while P_{T0} and P_{R0} are substitutions for terms that do not depend on the transmission range.

These equations were further reduced and used to consider multi-hop scenarios and to optimize media access protocols.

Other approaches like AEON [Landsiedel et al., 2004] considered real sensor applications, OS code and measurements to enable accurate prediction of the energy consumption. The work by Landsiedel et al. also mentions the idea of energy profiling. They achieved this by mapping source code functions to the corresponding object code addresses for TinyOS [Levis et al., 2005]. Since TinyOS is not written in a common programming language, but its own dialect of C, nesC [Gay et al., 2003], this approach is not portable to other platforms. Hence, this work does not provide a general approach to evaluate energy consumption in a real world deployment, but presents an interesting idea which is worth to be extended in a more generic way.

5.4.1.2 Simulation

To adapt the often complex models from theoretical analysis, simulation based studies typically use simplified assumptions. They usually describe a state machine and rely on basic formulas like equation 5.3 [Schmidt et al., 2007].

$$E = \sum_{state} P_{state} \cdot t_{state} + \sum_{trans} P_{trans} \cdot t_{trans} \tag{5.3}$$

Here P_{state} refers to the consumed power in *state* and t_{state} to the time spent in this state, while P_{trans} and t_{trans} have the same meaning for the transitions between two states.

Another approach, based on Final State Machines (FSMs), is proposed by Kellner et al. in [Kellner et al., 2008]. According to their work, evaluation of energy consumption using discrete event simulators is based on "less than adequate energy

[4] Digital Signal Processing

models". To overcome these shortcomings they attribute the states and transitions of the FSM with physical characteristics of the real hardware.

The authors of [Shnayder et al., 2004] propose another approach on simulating energy consumption. They describe PowerTOSSIM which they have developed as a scalable simulation environment for WSNs to provide accurate, per-node estimates of power consumption. A code-transformation technique is used to estimate the number of CPU cycles without the need for expensive instruction-level simulation. This makes this method feasible to study large scale networks. Since PowerTOSSIM is an extension to TOSSIM, an event-driven simulator for TinyOS applications, it is only suitable for this OS.

The simulation methodology for analyzing power consumption has two major drawbacks in general. The first shortcoming is that the formalized energy models often focus only on a particular network layer or application area, thus, limiting the generality of the experiments and results. The second problem is inherent to simulation for wireless multi-hop networks in general. As there is no real wireless medium but just a somehow simplified radio model, some effects that can be observed in the real world will potentially never happen in a simulation run. Besides, most simulators do not execute the same code as real hardware. Therefore, an error-prone translation from real code to simulation code is required.

5.4.1.3 Hardware Measurements

For reliable hardware measurements of energy consumption the difficulties are threefold. First, it can be hard to achieve accurate results and to prevent getting erroneous results due to misconfiguration or failure of the measurement devices. Second, one must think of a way to correlate the results to the actual program and software flow of the observed node. Third, measurement is often only feasible for a single node or at most for a small set of nodes in a lab setup, since measurement hardware is often expensive, needs manual configuration or administration, and is much harder to deploy to a large area then just deploying the sensor nodes.

The authors of [Kellner et al., 2008] introduce their own measurement platform SNMD (Sensor Node Management Device). By performing real hardware measurement on this device they try to validate and improve their FSM-based energy model. The core of measurement platform is an analog digital converter (ADC) that supports a resolution of down to approximately 76 µV and a frequency of up to 400 kHz.

Stathopoulos et al. introduces a novel architecture that consists of a hardware component and several software tools [Stathopoulos et al., 2008]. This work relies on an Application-Specific Integrated Circuit (ASIC) device that performs continuous real-time energy monitoring. As software tools they provide a kernel-space

energy measurement tool as well as a user-space observation tool that offers similar features like powertop for UNIX [powertop, 2011]. However, the ASIC itself requires up to 6 mW which is quite a lot compared to the typical consumption of IoT hardware. In comparison the receiving mode of the CC1100 radio transceiver consumes about 47 mW.

Another approach to estimate the energy consumption by hardware measurement is microbenchmarking. This term refers to a method that is used to measure the energy consumption of a very small piece of code or a particular component on the device. For this purpose a small program has to be designed that performs the function f_1 over and over again in a long running loop to later determine the average cost of a single execution of f_1. Microbenchmarks offer a practical method for analysis of power consumption when available measurement hardware does not support reasonable resolution.

A very promising work to analyze the power consumption of WSNs has been published 2008 by Fonseca et al.[Fonseca et al., 2008]. They adapted the before mentioned idea of energy profiling by Landsiedel et al. and combined it with their measuring tools as presented in [Dutta et al., 2008]. To track the power states of hardware, components they modified TinyOS's device drivers. In this manner, it is possible to correlate the observed current consumption to logical sets of operations, so-called *activities*. Although their approach is applicable to other platforms, there are two main difficulties to overcome. First, due to the event-based nature of TinyOS, the framework may not be easily ported to a thread-based OS like FreeRTOS or RIOT. Second, the implementation requires changes to a lot of components, particularly the device drivers. That implicates that some extra work is required every time a new driver is going to be implemented.

5.4.2 Current vs. Depletion Measurement

In order to measure the energy consumption of an embedded system two general approaches exist: **(i)** measuring *current* or **(ii)** measuring *electric charge*.

Current Measurement

For current measurements typically a measurement setup with a shunt resistor[5] is used. Measuring the voltage that drops across the shunt resistor allows to accurately compute the actual current value. The advantages of current measurement is the precise measurement of consumption peaks. This can be very useful for validating the correct behavior of software as well as for analyzing protocols. Analyzing the current over time can help to identify energy hotspots. The main limitations of

[5] A manganin resistor of accurately known resistance.

this approach is the limited sample rate of the setup. It can happen that temporal very small peeks with a high current are not sampled.

Electric Charge Measurement
In order to measure the electric charge (and thus derive battery depletion of an IoT system) typically a coulomb counter is used. The advantages of electric charge measurements are a very good accuracy for the overall energy consumption and a consequently accurate estimation of the realistic lifetime of a system. The disadvantage of this approach is the coarse resolution, that does not allow to get detailed information about the energy consumption of single functions or tasks of a program.

5.4.3 DES-eProf: Profiling Energy Consumption

The main idea of our methodology is borrowed from software engineering: *Software profiling*—as a tool for dynamic program analysis by instrumentation—enables, for example, code optimization. In software engineering, profiling is usually used to measure the usage of memory or frequency and duration of functions. This technique was adopted and extended it to allow for measuring the energy consumption of the system.

Three main components were used for the implementation of this method: **(i)** The GCC [GNU, 2016], **(ii)** RIOT, and **(iii)** a coulomb counter or a shunt resistor setup.

The GCC offers a compiler flag (*-finstrument-functions*) which allows to define functions to be called each time any function is entered or exited. By making use of this feature a detailed analysis of the function call tree is available. It can also create a detailed overview about how much time is spent in each single function. These *instrument functions* that are called at function entry and exit have to be very short running with little complexity. In order to use the GCC as compiler an OS written in C or C++, like RIOT, is required. The third component is the measurement platform itself which is going to be discussed in detail in Section 5.4.3.1.

In combination with some small modifications to the scheduler of RIOT, this enables a detailed analysis of the function call tree and enables per-thread evaluation of the energy consumption. By recording timestamps and energy consumption values, it can be exposed where most of the time and energy is spent within the program.

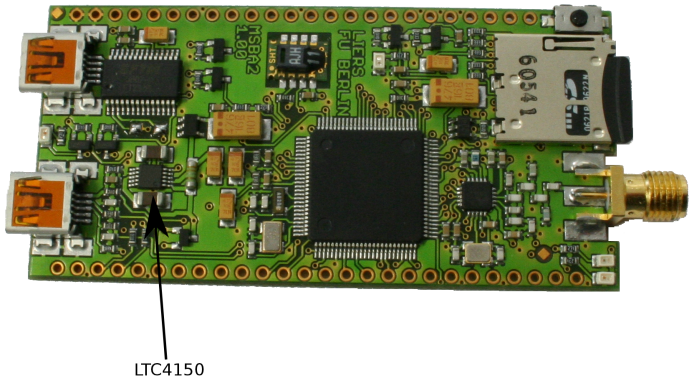


Figure 5.4: A MSB-A2 sensor node with LTC4150 coulomb counter.

5.4.3.1 The Measurement Instruments

The energy profiling tool was implemented on the MSB-A2 platform [Baar et al., 2008]. As shown in the picture Figure 5.4 this sensor node is equipped with a LTC4150 coulomb counter [Linear Technology, 2016]. This onboard device measures the electric charge consumed by the whole system. Due to the internal resistor and the maximum consumption of the whole sensor node, the smallest amount of electric charge to be measured is 93 mC (26 µA h). This corresponds to 85.8 µW at 3.3 V. Every time this amount of charge is consumed, a hardware interrupt is generated which can be handled by an interrupt service routine (ISR).

During the initial design and evaluation of this approach, it became clear that the resolution of the LTC4150 is too low [3]. A proposed solution to overcome this issue was to create a system of equations assuming that every function gets called often enough during an experiment to determine its energy profile. Unfortunately, it became evident that this solution would not work either, because the resolution of the LTC4150 was too low by several orders of magnitude. Hence, a new hardware measurement setup had to be designed.

This new measurement setup is realized by a small additional circuit which is integrated in the supply of the MSB-A2 and two additional wires for signaling. The first wire serves for the clock signal to keep the measuring device synchronous to the measured node. The second wire signalizes the transition to another state.

In order to carry out a differential measurement over a shunt resistor that could be integrated at any point of the supply line, we chose an instrumental amplifier to measure the voltage drop over the shunt resistor. As the MSB-A2 uses an internal clock of 72 MHz, we expect rapid changes in the current consumption because of the fast state changes in the MCU. Previous measurements under laboratory conditions had shown that the expected signal has a bandwidth of approximately

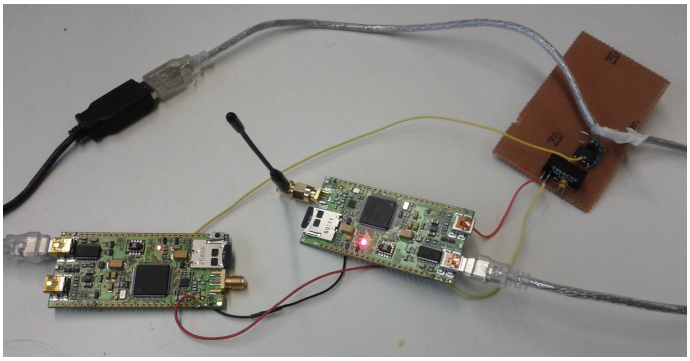


Figure 5.5: The measurement setup with the two MSB-A2s and the circuit comprising the shunt. The right MSB-A2 runs the program that is going to evaluated. The MSB-A2 on the left side reads the voltage from the shunt using its ADC.

200 kHz. Based on this analysis, we use a INA121P from Texas Instruments as an instrumentation amplifier [INA121, 2000]. This integrated circuit is suitable as it has a bandwidth with up to 600 kHz and is easy to use. The measurement circuit is implemented on an extra peripheral board with a connector for the power supply of the node we want to measure and an output which can be connected to the ADC of the measuring device. In our setup we use a second MSB-A2 as measuring device, which runs in parallel to the node we want to monitor as depicted in Figure 5.5. The described setup serves as a prototype for the next generation of MSB-A2 coulomb counter device.

The results from this circuit design were verified using a standard standalone laboratory oscilloscope. It turned out that the impact if the grounds of all partaking systems were not perfectly interconnected was initially underestimated. At the moment the measurements also include a lot of noise which needs to be reduced in future circuit improvements.

5.4.3.2 Measurement Results

As the described measurement device provides a maximum resolution of about 400 kHz the granularity of the profiling is adjustable. Hence, the energy profiling method is not bound to function-based or thread-based evaluation of the energy consumption but can evaluate any kind of functional block. However, the resolution is not high enough to monitor single MCU instructions. While the *instrument functions* records any transition between functions and the scheduler hook reveals information about thread switches, the measurement setup tracks the energy consumption. The recorded information about the energy consumption can be easily matched to the recorded data on the measured device itself due to the dedicated

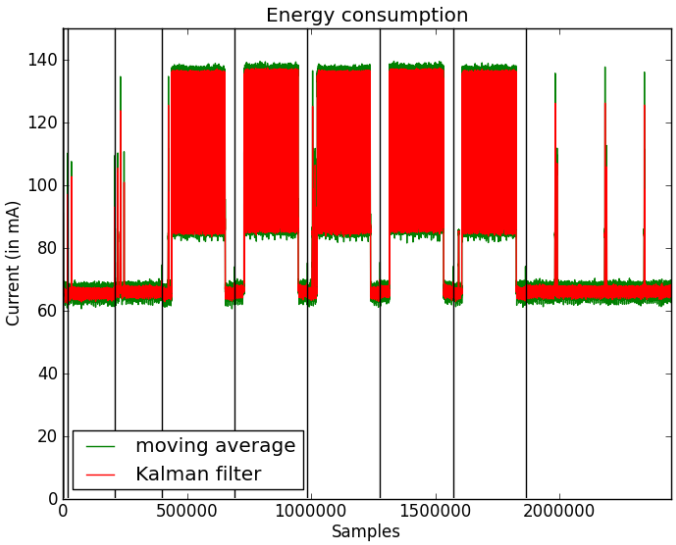


Figure 5.6: Measurement of an application using the CC1100 radio transceiver. Before the first vertical black line the kernel and timer gets initialized. Then the transceiver driver is initialized. After the next marker the transceiver goes to listening mode and starts sending out five broadcast packets after the next marker (delayed by 20 ms).

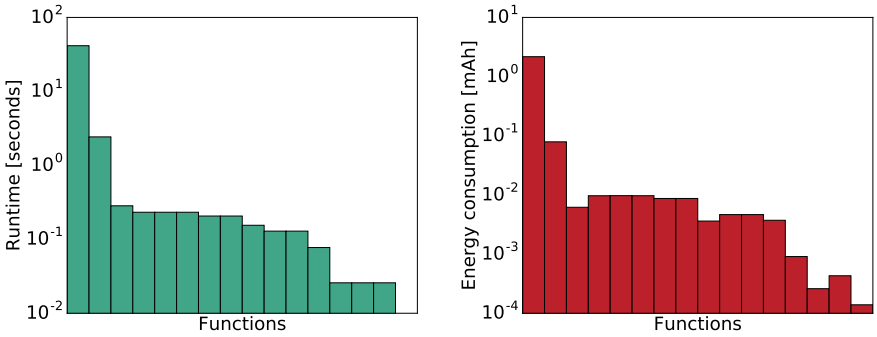
synchronization signalling between both devices. With the aid of the second signal line it is even possible to break the energy consumption down to any user-defined *activity*.

In order to store the data on the measurement device with the required data rate, we decided to use the integrated memory card[6]. The data (11 bit) is fetched from LPC2387's ADC with a frequency of 440 kHz and stored at first in 16 kB buffer within the RAM. This data than is written to the memory card using the DMA feature of the LPC2387 once the buffer is filled. From the flash memory card, the data can be easily evaluated directly on the PC.

We measured a basic WSN flooding application. Note, that low-power modes were deactivated during the experiments in order to allow shell access (see Section 4.1.3.

The results from the measurements are depicted in Figure 5.6. Due to the analogous character of the measuring setup the gathered data showed some noise. To smoothen this noise we tried a moving average and Kalman filter. The black, vertical markers are directly derived from the signaling wire.

[6]Using the serial output of the MSB-A2 was not feasible, because the used FTDI chip supports at most 3 Mbaud while the ADC of the LPC2387 provide a maximum bit rate of 4.4 Mbit/s [FTDI, 2010].



(a) Functions' runtimes. (b) Functions' energy consumption.

Figure 5.7: DES-eProf evaluation results on a per function basis.

The radio driver is configured to make use of the wake-on-radio (WOR) feature of the CC1100. Using this mode, the CC1100 scans the air only in defined intervals for incoming packets. Broadcast are sent using the burst mode in this configuration, to ensure that all receivers in transmission range will receive the packet. For the experiment the WOR interval was set to 542 ms. Furthermore, the stretched signal of the burst sending mode as well as the periodic listening of the transceiver is visible. The sending mode appears over the full 542 ms interval. In contrast, the periodic listening just generates a small peak in current consumption as no packet was recognized on the air and the transceiver returns immediately to power down mode.

Function Based Evaluation In Figure 5.7 the results from the DES-eProf on a per-function basis are shown[7]. The node run about half an hour sending a packet every 5 seconds. During the remaining time it was constantly switch to RX mode. It can be observed that the functions' runtimes do not necessarily correspond to their energy consumption—even though no lower modes were used. However, we note that not all energy consumed by components of the node is attributed to the proper function since some activities, e.g. the actual sending of the packets, happens asynchronously.

Per-thread Evaluation Listing 5.1 shows the results from the DES-eProf on a per-thread basis for a similar application. Additionally to the radio activity load was created by making extensive use of the serial communication interface to the sensor

[7]Since function names are rather meaningless without detailed knowledge about RIOT's port for the MSB-A2, they are omitted in the figure.

```
pid | name     | state     | pri | stack ( used) | runtime | consumption | switches
  0 | idle     | pending   |  15 |   500 (   96) | 86.341% | 76.423%     |     2355
  1 | main     | pending   |  14 |  2500 ( 1928) | 11.932% | 19.512%     |     2240
  2 | uart0    | bl rx     |  13 |  2500 (  224) |  0.810% |  0.813%     |     1052
  3 | cc1100   | bl rx     |   6 |  2500 (  288) |  0.205% |  2.439%     |      103
  4 | shell    | bl reply  |  13 |  4048 ( 1856) |  0.468% |  0.813%     |      358
    | SUM      |           |     | 12048
```

Listing 5.1: Results from a typical sensor application with flooding. The output of a defined terminal command *ps* is shown. By using this command, a user can query a per-thread statistical overview.

node. Since low-power modes were deactivated, the energy consumption of the *idle* thread is quite high.

5.4.4 Lessons Learned

A first insight is that large-scale experiments to measure the energy consumption are even more challenging than performance measurements. Not only that all LLN devices have to be equipped with an additional measurement device, but also the enormous amount of measurement data needs to be stored and processed.

A second insight: there is no one-fits-all solution for energy measurements, either. The researcher has to decide whether he/she wants to break down the energy consumption via current measurements or conduct a highly accurate, but less fine-grained depletion measurement via coulomb counter.

Finally, we can conclude that proposed energy profiling solution can indeed serve to verify—or disprove—existing energy models and can serve to provide realistic input parameters to these models. However, their general meaningfulness is limited and a measurement reveals rather insights about the measured hardware platform than about the software. Consequently, for an energy analysis of a networking protocol, it is advisable to use an accurate energy model and apply measurement data (e.g. from microbenchmarking) as input. Tools like DES-eProf are rather useful for debugging and reveal flaws in the implementation.

5.5 Summary and Contributions

One of the observations in part I was that experimental research is necessary for IoT. Therefore this chapter focusses on facilities and frameworks for experimental research on IoT and contributed tools in the domain of experiment-driven research on IoT.

Specifically, it analyzes two facilities in this area to conduct experiments: **(i)** testbeds and **(ii)** emulations. It studies the challenges, limitations, and requirements for these facilities and presents corresponding tools.

The facility that resembles real-world IoT deployments the closest is a testbed. A testbed allows the research to study algorithms and protocols in a realistic scenario, but also faces him/her with the challenge of deploying the software on the devices, conducting and controlling the experiment, monitoring the experiment's state as well as environmental parameters, and finally collect the results for evaluation.

In order to overcome some of the limitations of testbed-driven research, the use of emulators can be beneficial. Thus, this chapter presents the advantages and opportunities of hardware and network virtualization as emulation tools in IoT scenarios.

However, a property which cannot be easily examined with these common tools, is the fine-grained energy consumption of IoT applications. Hence, this chapter describes a concept to tackle this problem by combining software profiling techniques with a advanced energy measurement platform. The presentation of the different experimentation approaches is supplemented with a list of lessons learned that explain problems and short-comings of each of them.

Contributions

I was involved in the design and deployment of the DES-Testbed and development of the corresponding tools to facilitate testbed experimentation. The *DES-TBMS*, I co-developed together with the co-authors from [24] as a framework to reduce the burden on the researcher.

Moreover, in order to facilitate the application of the described virtualization tools, I extended the *DES-Virt* framework to support additional topologies, link-layer characteristics, and RIOT's *native* port.

I have designed DES-eProf, a new energy measurement tool for IoT devices, which can perform online energy measurements on a per-function or per-thread level. As a conclusion, I have observed that a tool like *DES-eProf* can be used to perceive a deeper understanding of the software's energy consumption, but leaves some open challenges in order to be deployed in large-scale.

The work in this chapter was published in the Proceedings of *IEEE International Conference on Internet of Things (iThings)* [4], the Proceedings of *GI/ITG KuVS Fachgespräch Sensornetze* [3], and the Proceedings of *IEEE International Conference on Communications (ICC)* [6].

Part III

Network Protocols for Energy Efficient and Reliable IoT

Chapter 6

An Information-centric Approach towards Energy Efficiency and Reliability over Low-Power and Lossy Links

Work presented in this chapter spawns from [18], which I co-authored with Emmanuel Baccelli, Christian Mehlis, Matthias Wählisch, and Thomas C. Schmidt.

Trade-offs between energy efficiency on the one hand and content availability or latency on the other hand, has proven to be challenging for the current IoT solutions as concluded in Chapter 2. This part will consequentially examine the possibilities to address these challenges using an information-centric approach. First, this chapter discusses the general opportunities, challenges, and potential benefits of ICN in the IoT.

Prior Work
Some prior work started to study ICN approaches for MANETs or other IoT-like scenarios via theoretical analysis and simulations. In [Biswas et al., 2013] and [Saadallah et al., 2012], preliminary tests are reported on small, toy networks. However, prior to this work there were no reports on larger scale deployments on IoT hardware, in environments matching requirements described in Chapter 1. Furthermore, prior work in this domain has either **(i)** focused on MANETs, where machines are typically not or much less constrained devices, or **(ii)** focused on WSNs and sink-centric data traffic (i.e. sensor-to-sink or sink-to-sensor) which is not representative of the whole IoT, where other types of devices participate, and other types of data traffic are significant, such as sensor-to-sensor traffic which is substantial in building automation scenarios (e.g., for lighting systems).

6.1 Why ICN for the IoT?

ICN was recently mentioned as a potential alternative networking solution for the IoT [Ghodsi et al., 2012]. It leverages in-network storage and multiparty communi-

cation through replication and interaction models such as publish-subscribe to provide efficient and reliable distribution of content. Using an ICN paradigm to interconnect IoT devices would provide a number of advantageous characteristics. For example, in-network caching enabled by ICN can reduce the amount of required radio transmissions, thus, reducing energy consumption and increasing local content availability while content producers are in power-save mode. Furthermore, an ICN paradigm could natively accommodate publish-subscribe traffic, which represents a large part of the expected IoT traffic. Last, but not least, by blurring the distinction between several mechanisms across layers, an ICN approach might **(i)** offer opportunities to efficiently factorize functionalities e.g., caching and buffering for error control **(ii)** drastically reduce the complexity of autoconfiguration mechanisms compared to an approach based on a layered protocol stack, **(iii)** achieve a smaller memory footprint compared to 6LoWPAN/IPv6/ RPL, and **(iv)** leverage the broadcast nature of the medium for enhanced caching and forwarding strategies.

While several ICN approaches have been developed, including NDN [Jacobson et al., 2009], PSIRP [Fotiou et al., 2012], Netinf [Dannewitz et al., 2013], DONA [Koponen et al., 2007], a number of key aspects remain challenges for ICN [Kutscher et al., 2016]. One example of such challenge is the design of routing schemes enabling automatic, efficient, and scalable forwarding information configuration on each ICN device. Related work proposed routing approaches based on proactive, link-state mechanisms [Hoque et al., 2013, Wang et al., 2012]. However, such approaches may not be directly applicable in the IoT, where constrained devices impose different requirements in terms of memory and power capacities.

6.2 Challenges for ICN in LLNs

The usage of ICN solutions in IoT deployments display many advantageous properties, but leave some open challenges. Some properties of IoT scenarios do not naturally match the design of ICN networks. This section lists some of these challenges.

In the following, we will assume NDN is the concrete base upon which we build our information-centric IoT model, which we run directly above the MAC layer.

6.2.1 Link Layer Considerations

With NDN, a *face* is an abstraction that maps to a physical network interface, e.g. a Network Interface Controller (NIC) to a point-to-point link, or a logical interface, e.g. an interface to the application. In IoT scenarios however, mapping a *face* to a

NIC may not be appropriate since nodes are typically equipped with a single NIC connecting to a wireless broadcast medium. Ideally a face can be mapped to a link layer address in such a way that the MCU does not have to be switched on if a packet that cannot be handled is received. Hence, three different approaches to address this issue are proposed:

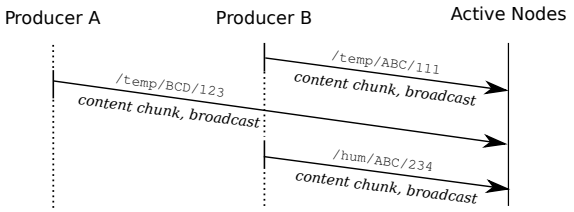
Static and Dynamic Faces Departing from basic NDN, we distinguish between two types of faces: *static faces* and rather short-lived, *dynamic faces*. Static faces are used for broadcast and loopback communication (the loopback face is also the interface to the application). Dynamic faces are bound to the destination address of the peer in the convergence layer, in our case, a link-layer address. A dynamic face is linked to current PIT entries, and the face is removed if no PIT entry refers to it anymore.

Overhearing Alternatively ICN could try to leverage the broadcast nature of the medium at the best. In this approach all PIT and FIB entries are mapped to a single *broadcast face*. Consequently, RPF is also done in a broadcast fashion. This approach can be beneficial if combined with *gossip*-like forwarding [Angius et al., 2012, Blywis et al., 2010b] in deployments with a high degree of asymmetrical or unidirectional links.

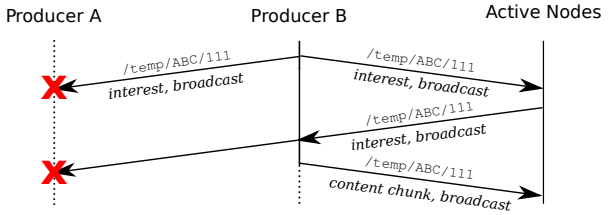
Non-Broadcast Multi-Access Some LLN link layers (compare Section 2.3.1) provide coordinated medium access in such a way that the broadcast medium is divided into temporal slots and/or radio channels which are dedicated to a certain link. With such a link layer ICN *faces* can be directly mapped to a particular neighbor node. For that, a tight coupling with the scheduling sublayer is required.

6.2.2 Autoconfigured Names

Since manual configuration and management of network properties is impracticable for (large scale) IoT deployments, mechanisms for autoconfiguration needs to be in place. For NDN, this implies in particular that names must be autoconfigured at bootstrap. We thus enhance NDN with a name autoconfiguration scheme. Each name must satisfy the requirements of **(i)** meaningfulness and **(ii)** uniqueness. In order to satisfy these requirements, we assume a prefix that consists of sensor type identifier and a unique identifier of the node in the name, e.g. a vendor ID. Additionally, we extend the prefix of the name by a suffix, the timestamp, which can also serve as a version number. The name could be enhanced by further information, e.g., based on geographical or organizational properties. A name generated by the



(a) Distributing data from sensors to neighbors as unsolicited content chunk.



(b) Distributing data from sensors to neighbors using Interest-Interest.

Figure 6.1: Basic communication schemes for IoT push traffic.

autoconfiguration mechanism looks like */temp/DEADBEEF/*1466250645. We observe that the three components of this name (*sensor type*, *node ID*, and *timestamp*) can be derived locally on the node.

6.2.3 Support of Push Traffic

Natively, NDN does not allow unsolicited content transmission. A node that receives unsolicited content should discard this chunk. This restriction makes it difficult to allow *push traffic*.

Interest-Interest One approach to enable *push traffic* with NDN is proposed as *Interest-Interest* approach to control actuators [Burke et al., 2013]. The node that wants to push data, expresses an Interest that prompts the consumer to express an Interest for the data that should be pushed itself.

Permanent PIT Entries Another approach that enables the transmission of periodical or event-triggered content, as it is common in sensor networks, is proposed in [Amadeo et al., 2014a], allowing non-routable unsolicited data for single-hop scenarios. We use a similar approach to allow *producers* to locally broadcast unsolicited content. Active nodes have a permanent PIT entry matching the wildcard prefix */∗* and hence, do not discard unsolicited content chunks.

Encapsulation into Names Finally, small content could be also included in the Interest as a name component [Amadeo et al., 2014a]. NDN does not pose any restrictions on the namespace or the number of name components. Using a (dummy) data packet that is sent in response to this content encapsulating Interest can serve as an implicit acknowledgement.

6.2.4 Asymmetric and Unidirectional Links

Many ICN approaches assume bidirectional links. This is not true in general in spontaneous wireless networks [Cordero et al., 2013], and thus this assumption does not hold in the IoT. In such context, a high proportion of links are asymmetric, e.g., 10 % loss rate from A to B and 80 % loss rate from B to A. In reality, a substantial fraction of the links are unidirectional, i.e. loss rate strictly below 100 % in one direction, and 100 % loss rate in the reverse direction. Last but not least, wireless link quality between two nodes A and B can vary significantly over time, even at small time scales [Baccelli and Perkins, 2016]—a phenomenon we also experienced in our experiments.

The above wireless connectivity characteristics lead to the following observations. ICN routing protocols running on constrained devices need to satisfy conflicting requirements: **(i)** negligible control traffic to reduce energy consumption and small state to fit memory constraints, while at the same time **(ii)** dynamic tracking of wireless link to avoid non-functional paths. The goal is to not forward an Interest in the first place if reverse link is not "good enough". The overhead for failing is a reverse path taken by content which often fails and will lead to PIT time-outs, Interest flooding, etc. Subsequently, this might lead to the same failing reverse path—and thus be very inefficient both in terms of energy and delay.

6.3 Routing in ICN IoT Scenarios

Reduced memory of constrained devices limits applicability of ICN routing approaches. Current proposals usually route either directly on names or indirectly via name resolution. Based on our previous observations, name resolution on top of IP is not advisable. However, even some pure name-based routing schemes, such as [Hoque et al., 2013] and [Wang et al., 2012] rely on an ICN overlay requiring an IP network, or use proactive link state algorithms. Link state routing results in both, **(i)** a significant amount of control traffic, whether or not there is data traffic to carry in the network, and **(ii)** a significant amount of memory, typically in $O(n)$, where n is the number of nodes in the network. These characteristics do not match the memory and energy resources of constrained devices. Routing protocols run-

ning on IoT devices should aim for O(1) routing state and minimal control traffic—ideally none, especially when there is no data traffic to carry [Levis et al., 2009]. In the following we propose and evaluate such a mechanism.

6.3.1 Basic Routing Mechanisms for Information-Centric IoT

Vanilla Interest Flooding (VIF) The simplest routing approach that requires minimal states is *Interest flooding*, whereby each node in the network repeats an Interest, upon first reception. In the following, we will call this simple mechanism VIF. Using VIF, a consumer with an empty FIB can nevertheless disseminate its Interest in content, and the flooded Interest will reach the producer which can then send the content on the reverse path. VIF fits the constraints of IoT devices because (i) it does not rely on any additional control traffic to maintain the FIB, (ii) it requires minimal state, i.e. only temporary pending Interests on the reverse path of content that is sought after.

While experimental evaluation was successful in that NDN was demonstrated to operate on IoT hardware (meeting memory requirements), and the consumer could fetch the content, we observed that, compared to its size, many packets were transmitted to fetch the content. This is due to the fact that each chunk triggers an Interest, which requires network-wide flooding. The results of this experiment are presented in Section 6.3.2.

In general, in a network of n nodes, and for k chunks of content, the number of transmissions for a single content item is $k \cdot ((n-1) + \sqrt{n})$, assuming the average path length approximation $\sqrt{n}$. We observe that while VIF is simple and works in the scenario we tested, it does not scale well in terms of number radio transmissions when the network or the content grows in size. Radio transmission and reception are however very costly in terms of energy for battery-powered IoT devices. In the following, we have thus designed and tested enhancements reducing the number of radio transmissions and receptions in IoT environment.

Reactive Optimistic Name-based Routing (RONR) In order to reduce the number of radio transmissions compared to basic Interest flooding, we introduce RONR, which automatically configures a temporary FIB entry on the reverse path taken by the first content chunk. That way, in case the FIB is empty (e.g., after booting) or if no FIB entry matches the name/prefix of the content in which the consumer is interested, only a single initial Interest flooding is needed, while subsequent Interests for chunks of that content can be unicast using the FIB entries thus auto-configured along the path. For example, in our experiments, after flooding an Interest for chunk */riot/text/a*, nodes on the reverse path of that chunk store

a temporary FIB entry for */riot/text/**, thus subsequent Interests for chunks */riot/text/b*, */riot/text/c* can be unicast using the established path, instead of flooded.

RONR is optimistic because it first assumes that the whole content is stored on a single node (a cached replica or the original producer), which may not be the case in general. However, this assumption is reasonable in the IoT because typical content size is in the order of a few hundred megabytes [Martocci et al., 2010]. Furthermore, FIB entries timeout ensure that if the configured FIB entries do not lead to a node with the full content, the consumer will eventually revert to Interest flooding, through which it can discover another node with the rest of the content, install new temporary FIB entries etc. This timeout strategy is common for reactive routing in multi-hop wireless scenarios [Richard et al., 2005].

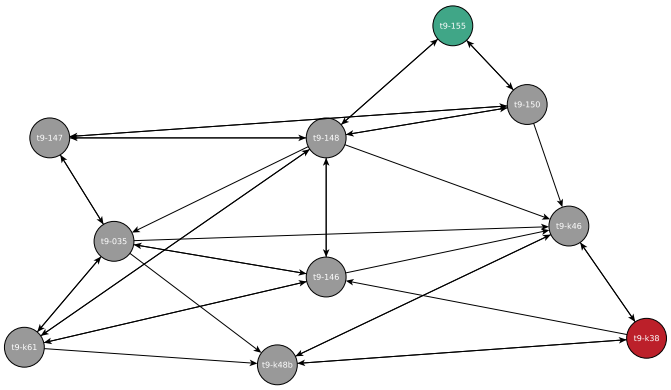
6.3.2 Experimental Evaluation

In order to evaluate the proposed routing schemes, experiments in a testbed were conducted. We will first describe the basic experiment configuration we used for this and following evaluations and then present the results.

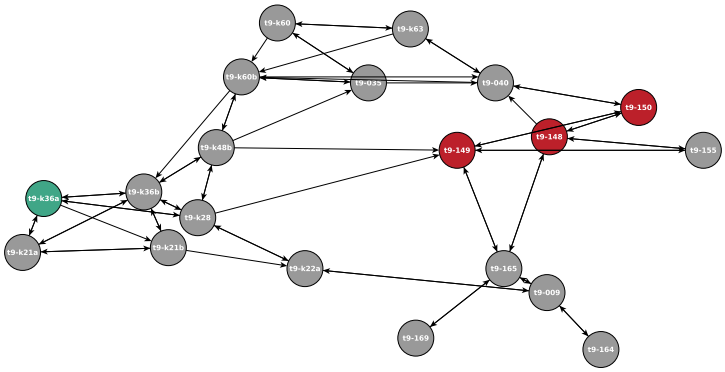
Basic Configuration of Experiments

For the experiments in this and the following sections, we used the DES-Testbed. The following experiments use 400 ms Interest timeout (stop-and-go, giving up after 5 tries), and 900 ms nonce timeouts. The content is named in a hierarchical fashion typical for NDN. Encryption was disabled for the experiments. Considering the maximum link layer frame size of 64 byte in our deployment, we decide for a medium sized name length of 12 byte including the chunk identifier (the exact names of the content chunks are */riot/text/a*, */riot/text/b* etc.). Note, that with these names, the size of headers and names fit in a single link layer frame, both with CCN (16+12 = 28 byte) and with 6LoWPAN/ RPL/ UDP (15+12 = 27 byte), and still allow to carry realistic application data. Also note that the sizes of minimal CCN header (16 byte, eliding optional fields) and of 6LoWPAN/ RPL/ UDP headers (15 byte) are similar, and thus represent not a decisive factor in the differences observed in the following experiments.

For our experiments, we considered the use case of a HVAC system as described in Section 1.2. In the experiments, we consider a single content producer and one or multiple consumers. Due to the volatile nature of the wireless medium [Baccelli and Perkins, 2016], the resulting link layer topologies based on our 60 node network might change on a per-transmission basis (cf., Figure 6.2). Note that IoT scenarios in home and building automation networks are typically multi-hop, but less than 5 hops in diameter [Martocci et al., 2013]. Consequently, in our experiments, we placed content producer and consumers at least 2 hops apart.



(a) 10 nodes are involved when a single consumer (red colored) requests content published by green node.

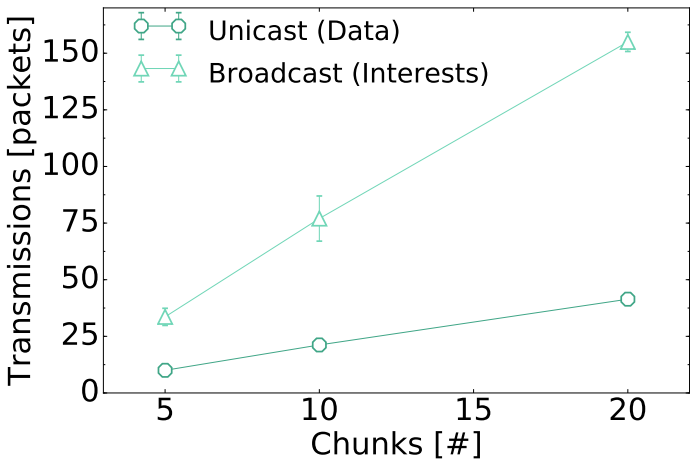


(b) 20 nodes are involved when multiple consumers (red colored) request content published by the green node

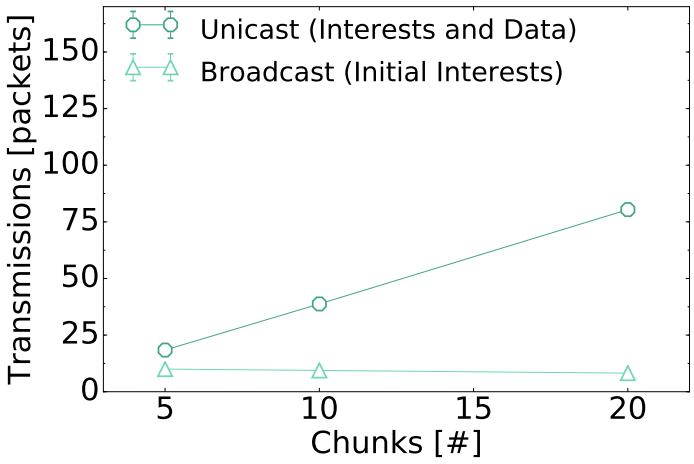
Figure 6.2: Snapshot of the link-layer network topologies used in the experiments for single and multi consumer scenarios. Link weights describe % of received packets, per link, per direction.

To analyze the effects of NDN for typical radio packets payload in the IoT, we align the chunk size such that each chunk can be transmitted without fragmentation. In our case, MTU is 64 byte, chunks are set to be 58 byte long, of which 30 byte of content. Since typical sensor content production in HVAC deployments is of the order of 200 byte per minute [Martocci et al., 2010], we set the basic configuration for consumers to periodically fetch 10 such chunks. In order to compare to other IoT radio technologies with bigger or smaller MTUs into account, we also examined cases with 5 and 20 chunks per content item.

Figure 6.3(a) shows the results of an experiment using NDN with VIF for a single consumer scenario. In this experiment, the consumer periodically accesses content of size 5, 10, or 20 chunks of data, all of which were produced by another



(a) Vanilla Interest Flooding



(b) Reactive Optimistic Name-based Routing

Figure 6.3: Single-consumer scenario. NDN performance for different routing schemes. Average number of packets transmitted in a network of 10 nodes to fetch content of various size.

constrained node in the network shown in Figure 6.2(a). To better understand the nature of the network traffic, we distinguish broadcast packets (which are Interest packets in this case), from unicast packets (which are content in this case).

In Figure 6.3(b), we show the results of an experiment using NDN with RONR, for the exact same topology and scenario as for Figure 6.3(a). We observe that the number of radio transmissions decrease about 50 % compared to NDN with VIF. In particular the number of broadcast transmissions is drastically reduced because, with RONR, only the first Interest packet of a content item is flooded, while sub-

sequent Interests are unicast, using temporary FIB entries established by RONR. A quick back-of-the-envelope analysis shows that in a network of n nodes, and for k chunks of content, the number of transmissions is $(n-1) + 2(k - \frac{1}{2})\sqrt{n}$, assuming again the average path length approximation $\sqrt{n}$. Therefore, RONR scales much better than VIF when network size or content size grows. RONR thus better fits IoT devices energy requirements compared to VIF, while still fitting other requirements of constrained devices by (i) not relying on any control traffic, and (ii) requiring minimal state, i.e. only temporary FIB entries on the reverse path of content that is sought after (not counting PIT state, of course).

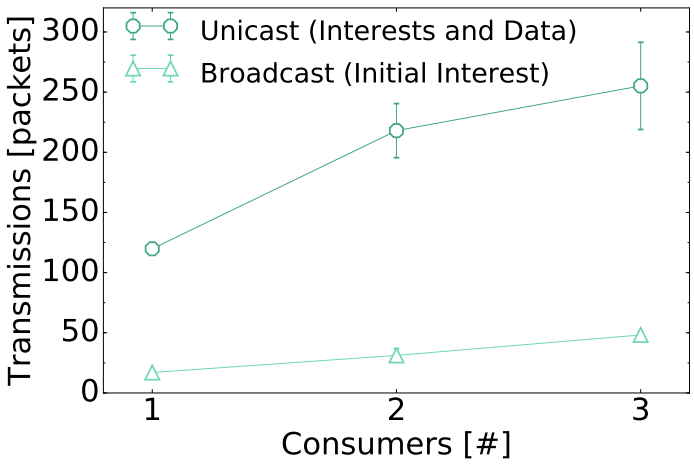
6.4 Multiple Consumers & Impact of Caching

In some IoT scenarios several consumers may be interested in the same content. For example, several devices in a HVAC network could want to access the most recent data generated by a sensor e.g., a temperature sensor asynchronously accessed by the air-conditioning system, the automated blinds, and windows of a room, each of which may react independently upon temperature evolution. In these scenarios, ICN's in-network caching abilities are advantageous in several aspects. They reduce **(i)** the average time to fetch this content for the consumer, **(ii)** the overall traffic load of the network, and **(iii)** the energy consumption per node..

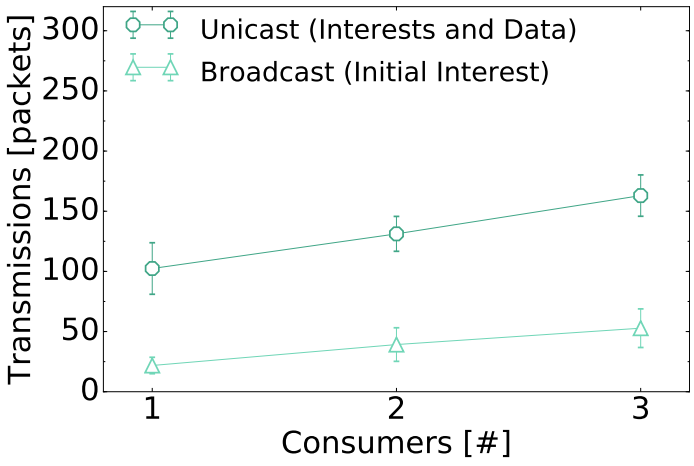
We evaluate experimentally the impact of ICN caching in such a scenario. The same content (20 chunks) as in the previous experiments is accessed alternatively by one, two, or three consumers that are topologically close to one another (pairwise, maximum hop distance is 1). In order to accommodate for more consumers while keeping them apart from the producer with at least 2 hops, a larger topology shown in Figure 6.2(b) was used for the following experiments. To reduce signaling overhead, we use RONR as routing scheme for NDN interest packets.

In Figure 6.4(a) we show the results of our experiment with a disabled content cache. We observe that, as expected, the number of radio transmissions scales almost linearly with the number of consumers. In a network of n nodes, and for k chunks of content and m consumers within radio reach, the number of transmissions is $m \cdot ((n-1) + 2(k - \frac{1}{2})\sqrt{n})$, still assuming the average path length approximation $\sqrt{n}$.

Next, we enable cache capacity of 20 chunks on all nodes, which corresponds to RAM usage of 2 kB (2 % of 96 kB overall RAM). Figure 6.4(b) shows the results we obtained for the exact same topology and scenario as for Figure 6.4(a), except the caching. We observe that the number of radio transmissions needed to retrieve the content is drastically reduced, by up to 50 % in this scenario. In detail, the number of broadcast transmissions is almost similar, while the number of unicast



(a) Without caching



(b) With caching

Figure 6.4: Multi-consumer scenario. NDN performance for RONR and different content cache schemes. Average number of packets transmitted in a network of 20 nodes with a variable number of consumers.

packets decreases substantially. This is consistent with the facts that the initial interest flooding (broadcasted) is not modified, while cached content chunk shorten unicast paths, thus reducing the number of unicast transmissions. In the best case, if the initial flood for subsequent consumers can be reduced to a local broadcast because only neighbors with cached content receive the interest, the number of transmissions becomes $2(k - \frac{1}{2})(\sqrt{n} + n - 1) + n + m - 2$.

6.5 Comparison to 6LoWPAN

In order to compare the standard IoT network stack as presented in Section 2.3 to an NDN IoT solution, a qualitative comparison of the required mechanisms for large-scale and energy-efficient IoT scenarios as well as a quantitative comparison are required.

6.5.1 A Qualitative Comparison

For a qualitative comparison with IP-based IoT network protocols, an analysis of ICN mechanisms addressing the requirements of Smart Object networks at several layers is required. The following analysis will compare these mechanisms to the according ones from the standard IP-based IoT network stack as described in Sections 2.3 and 2.4.

6.5.1.1 Core Mechanisms

Content Aware On the content aware layer NDN follows a basic publish-subscribe model [Eugster et al., 2003], which CoAP addresses via extensions such as the observe mode or the publish-subscribe broker. In contrast to observe mode, NDN does not natively support push traffic for updated content chunks. Corresponding extensions for NDN in the local scope of LLNs are available as discussed in Section 6.2.3. In NDN, however, there is no need for an explicit publish-subscribe broker as its routing mechanisms and in-network caching capabilities provide that already.

Since there is no host-centric communication model in NDN, there is also no need for a dedicated proxy that caches content from sleepy nodes. Each node in a NDN network can replicate content generated by other nodes in the network.

NDN's publish-subscribe communication paradigm is also well-suited for service discovery. Naming conventions can help to discover device capabilities, while in-network caching can help to disseminate this information with a low amount of traffic.

Transport NDN is "designed to operate on top of unreliable packet delivery services" [Jacobson et al., 2009] which makes it very appealing for the IoT. Its implicit ACK mechanism triggers Interest retransmissions automatically and thus, does not require explicit mechanisms like confirmable messages in CoAP.

NDN also provide basic flow control mechanisms, where Interests serve the role of window advertisement. The flow balance in NDN is maintained at each hop and all communication is local [Jacobson et al., 2009]. However, additional congestion

control mechanisms are required to prevent Denial of Service (DoS) attacks through Interest storms [Wählisch et al., 2013].

For segmentation, content is divided in so-called *chunks* in NDN. Hence, a content producer that is aware of the particular transport's MTU, can decide to generate chunks that obey the available payload size, e.g. of a link layer frame[1]. Each content chunk must be requested by a corresponding Interest packet. This matches very closely block-wise transfer in CoAP. As in CoAP block-wise transfer the server/producer is not required to keep any state. Moreover, content chunk numbers ensure the right order of content delivery, which also corresponds to typical service of a reliable transport mechanism.

Routing Many of the mechanisms used for IP-based routing in LLNs described in Section 2.3.3, like energy-aware metrics, are still applicable in NDN IoT deployments. In contrast to IP-based IoT networks however, routing for NDN IoT scenarios is directly based on names instead of locators (cf. Section 6.3). As a consequence, the naming scheme has a big impact on the routing performance and resource requirements. While a flat name space is often desirable in LLNs because of the reduced name lengths, it poses a challenge to the routing protocol due to lacking aggregation possibilities. In general, limited aggregation potentials pose one of the major challenges for name-based routing [Schmidt et al., 2016]. In many IoT scenarios, the gateway may take the role of the root node of the routing tree and serve as an aggregation point.

Network On the network layer there is no need for potentially traffic intense mechanisms like IPv6 ND, since NDN nodes are not interested in their neighbors' addresses, but only in their contents. Similar, to the bootstrapping in 6LoWPAN networks however, NDN needs to autoconfigure its names This requires a naming authority, similar to an authoritative 6lbr in 6LoWPAN and may even require some equivalent to duplicate address detection in IPv6 (some kind of *duplicate* name *detection*).

The header in NDN is typically considerably smaller than in IPv6, but the length of the contents' names have a significant impact on this. Hence, it can be considered to deploy a corresponding mechanism to 6LoWPAN's header compression for NDN names.

Another task of this layer is the transport of packets that do not fit into a single link layer frame. In the IP-stack a packet can be fragmented in a hop-by-hop manner using either the 6LoWPAN or IPv6 fragmentation. Comparably NDN supports

[1]However, chunks cannot get arbitrarily small. In fact, further work on name and header compression mechanisms is required to achieve reasonable small chunk sizes.

hop-by-hop fragmentation [Mosko and Tschudin, 2016]. The advantage of fragmentation over splitting the content in more chunks on the transport layer is the reduced number of Interest packets, because each content chunk must be requested by a corresponding Interest packet. The disadvantage is a processing overhead, since the fragmented packet needs to be reassembled and fragmented again by each intermediate hop. Thus, NDN's fragmentation is very similar to fragmentation in 6LoWPAN.

Medium Access Since NDN works always above the link-layer, it has obviously no direct impact on Medium Access mechanisms and can operate over the same protocols described as in Section 2.3.1. However, clever mapping of names to faces as discussed in Section 6.2.1 can greatly help to improve the energy efficiency. We will discuss further advantages of a tight coupling between NDN and the link layer in Chapter 8.

6.5.1.2 Auxiliary Mechanisms

Security The standard 6LoWPAN stack comprises security mechanisms on various layers as seen in Section 2.4.1. These mechanisms range from cryptographically protected application payloads to encrypted link-layer transmissions and focus typically on securing the channel [Heer et al., 2011]. In contrast to this concept, NDN focusses on securing the data itself. In this manner, no secured end-to-end connection is required and data can traverse boundaries between heterogeneous network environments in a secure fashion. However, while application data is always signed at the time of production and can additionally be encrypted, an attacker may still be able to disclose the communication partners and meta-data derived from the content's name. Hence, additional mechanisms, like encrypted transport (e.g. IEEE 802.15.4 security) are still required for NDN to conserve privacy. Moreover, the visibility of names to all forwarders, non-ephemeral keys, and opening of the control plane introduce many new challenges to security in NDN [Wählisch et al., 2013].

Network Management While several mechanisms—many of them based on CoAP—exist for network management in the standard 6LoWPAN stack as presented in Section 2.4.2, not much work on this particular topic has been conducted for NDN IoT so far. The original NDN project considers management procedures only for storage and usable trust. The authors of [Corujo et al., 2012] propose a management framework for NDN using manager entities and management agents for management coordination. However, this approach seems to be too heavy-weight for IoT. Hence, more work in this area is required.

6.5.2 A Quantitative Comparison

Next, an analysis of NDN IoT to the IP-based approach is required in terms of **(i)** memory requirements and **(ii)** traffic overhead.

Memory Requirements

The protocol state machine of the NDN stack is less complex compared to the full 6LoWPAN stack. As a consequence, less memory is required for the implementation, both, in terms of flash and RAM usage. The conserved memory can be leveraged for content caching.

Table 6.1 compares the ROM and RAM sizes of the binaries compiled for NDN/CCN network stacks and for 6LoWPAN network stacks[2], built upon state-of-the-art IoT OSs (RIOT and Contiki), for comparable IoT hardware (Redbee Econotag board and MSB-A2 board). Only the memory consumption of the upper layers of the stack are compared. For RIOT the NDN implementation of CCN-Lite [CCN-Lite, 2014] is used. For Contiki a CCNx implementation has been evaluated [Saadallah et al., 2012]. We observe that an NDN approach can significantly outperform common IoT protocols in terms of ROM size (down to 80 % less) and RAM size (down to 65 % less).

Traffic Evaluation

Next, NDN is compared to the 6LoWPAN stack with respect to induced (control) traffic requirements. For fair comparison, we use the following setup: On the NDN side, we deploy RONR with a cache size of 2 kB, as this leads to the best performance results in our previous analysis. On the RPL side, we first let the network converge until the RPL root and the routing entries are installed in nodes, before we start the experiment (i.e. we factor out the control traffic transmissions necessary to bootstrap the network).

In Figure 6.5, we show the results we obtained for the exact same topology and scenario as for Figure 6.4(b), except the network stack used was 6LoWPAN stack with default settings instead of NDN. We observe that the 6LoWPAN network stack yields much more transmissions compared to NDN (cf., Figure 6.4(b)), approximately three times more. This is due to two main factors. On one hand, the amount of control traffic generated by the *proactive* 6LoWPAN network stack is a big penalty compared to the *reactive* NDN approach we tested. On the other hand, compared to our NDN approach, the unicast paths created by the 6LoWPAN network stack do not benefit from *caching* and are thus always maximum length, which

[2]The considered 6LoWPAN stack application used for the comparison in this section comprises also CoAP, UDP, and RPL

(a) RIOT on IoT-LAB-M3

Module	ROM	RAM
CoAP + RPL + 6LoWPAN	48 491 byte	10 754 byte
NDN	15 614 byte	2767 byte

(b) RIOT on MSB-A2

Module	ROM	RAM
CoAP + RPL + 6LoWPAN	78 617 byte	8834 byte
NDN	22 206 byte	3551 byte

(c) Contiki on Redbee-Econotag

Module	ROM	RAM
CoAP + RPL + 6LoWPAN	61 371 byte	16 520 byte
CCN	13 005 byte	5709 byte

Table 6.1: Comparing memory resources for common IoT operating systems and hardware.

can in some cases be even longer than the shortest topological paths, as shown in [Xie et al., 2010]. Note that we have not used RPL extensions such as reactive point-to-point route discovery [Martocci et al., 2013], which could reduce the length of unicast paths. Furthermore, as discussed in Section 6.3, we observed that the naming scheme and the header sizes were not a decisive factor explaining the performance gap between the NDN stack and the 6LoWPAN stack in the experiments we conducted. All in all, we can conclude that NDN may be a potential alternative to 6LoWPAN, which should be studied more in the context of IoT in future work.

6.6 Summary and Contributions

The analysis of the state of the art in Part I revealed that the current approach towards energy efficient and reliable communication in IoT use cases leaves some open challenges. This chapter proposes to consider information-centric approaches in order to address some of these challenges. Consequently, it studies the applicability of ICN for IoT scenarios and identifies a set of challenges. A name-based routing protocol is proposed and evaluated on testbed-based experiments. The last part of this chapter discusses how NDN approaches compare semantically as well as in terms of memory consumption and required traffic overhead to the standard 6LoWPAN approach.

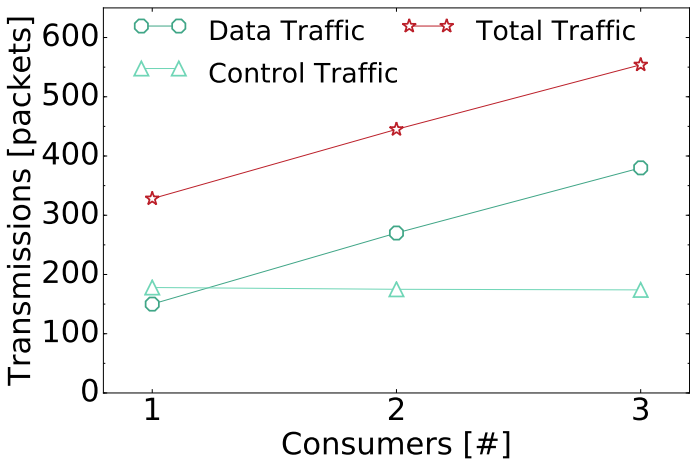


Figure 6.5: Multi-consumer scenario with 6LoWPAN. Average number of packets transmitted in a network of 20 nodes.

Contributions

I have provided one of the first experimental comparative evaluations of ICN for the IoT with the standard 6LoWPAN stack. Based on extensive experimental studies I have shown that ICN is applicable for low-end IoT devices in the LLNs, and that it can offer advantages over an approach based on the standard 6LoWPAN stack in terms of energy consumption, as well as in terms of RAM and ROM footprint. Together with the co-authors from [18] I have proposed several interoperable NDN enhancements to decrease energy consumption and routing state. I have co-developed RONR as a routing protocol for NDN with low complexity and minimal memory footprint. Compared to a 6LoWPAN solution the overall memory consumption is reduced by a factor of 4 or 5 and traffic in multi-consumer scenarios can be reduced by up to 66 %.

The work in this chapter was published in the *ACM conference on Information-centric Networking (ICN)* [18].

Chapter 7

Information-Centric Cooperative Caching Strategies for Energy Efficiency

Work presented in this chapter spawns from [14], which I co-authored with Emmanuel Baccelli, Matthias Wählisch, Thomas C. Schmidt, and Cedric Adjih.

Exploiting the in-network caching abilities of the information-centric paradigm can be used to improve on the energy trade-offs in the IoT described in Section 2.6. This chapter particularly addresses the trade-off between energy efficiency and content availability. It studies how ICN and name-based caching strategies can help to improve on energy efficiency without decreasing the level of content availability in IoT scenarios.

Prior Work
In [Amadeo et al., 2014b] a NDN optimisation is proposed to exploit the wireless broadcast nature of IoT networks to retrieve content from multiple producers with a single interest, using persistent PIT entries. In [Quevedo et al., 2014] authors propose complementary mechanisms to adapt NDN to information freshness requirements specific to IoT sensor data. The authors of [Hail et al., 2015] examine forwarding and caching in wireless IoT systems using a simulator. Among others they study a basic random caching strategy with LRU and observe performance gains in content delivery, via simulations on a grid topology. This chapter includes the first analysis on distributed caching strategies in IoT on real hardware, that addresses the trade-off of data availability and energy efficiency.

7.1 Information-centric Support for Sleeping Nodes

In some IoT use cases, no designated gateway or proxy is available most of the time. One example is *in-the-wild monitoring of plants, soils, or animals*, which requires a large number of small IoT devices embarking sensors, disseminated in an area, e.g. on a meadow.

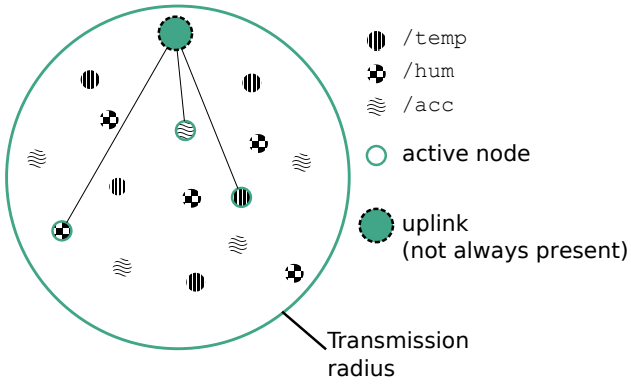


Figure 7.1: Broadcast domain.

Another example is *monitoring large storage locations*, which requires a large number of scattered IoT devices with sensors tracking the state of monitored goods or machines. In such use cases, there is no designated proxy and the cloud is reachable infrequently, e.g., only when a drone with appropriate communication capabilities flies by or when an employee tours to check the area, carrying a tablet that polls sensors via radio communications.

We base the following discussions on a basic IoT scenario: a single wireless broadcast domain that gathers a set of sensors of various types as shown in Fig. 7.1. This domain is connected to the Internet via an intermittent uplink. When the uplink turns up, all active nodes transmit their cached data to the uplink.

We consider that wireless links between nodes have similar capacity, but are subject to constraints of a LLN as discussed in Section 2.1, i.e. low capacity, low processing power of MCUs, low data rates, and a high probability for interferences. These assumptions fit rather well both dense IoT deployment in industrial environments, and wildlife area monitoring environments using lower radio frequencies and less encumbered, line-of-sight radio communication.

In detail, each sensor is a content producer, and is hosted on an IoT device as shown in Fig. 7.2. Each IoT device provides a small cache (RAM $\approx$ 50 kB) and a low-power CPU, to which are connected peripherals including a low-power radio interface.

We consider that the sources of IoT data are sensors, which each generate data as time series of sensor readings. We assume that each chunk of content is big enough to contain one sensor reading, and small enough to fit into a single radio packet transmission. We focus on scenarios where sensors monitor a phenomenon whereby (i) data relevance strictly decreases with time, and (ii) a more complete view of what the sensors are monitoring is achieved if available data comes from a larger number of distinct sources (i.e. sensors).

We then evaluate the availability of IoT content considering the below two metrics:

Diversity corresponds to the requirement to have a complete view of what the sensors are monitoring. Maximum diversity is achieved if content from all possible sources is retrieved by the uplink.

Freshness corresponds to the requirement to have an up-to-date view of what the sensors are monitoring. Maximum freshness is achieved if the newest data is retrieved by the uplink.

7.2 Sleeping & Caching Strategies

Our goal of deploying NDN in the IoT is to improve energy efficiency while maintaining availability of recent data. The core questions that need to be addressed are **(i)** how to organize sleeping of nodes to best use scarce energy resources, and **(ii)** how to organize cache maintenance as memory is limited. We define and analyze a number of approaches for sleeping and caching.

7.2.1 Sleeping Strategies

Nodes alternate between *active* and *sleeping* phase according to a sleeping strategy, which can be either coordinated or uncoordinated. Upon generation of new content, a sensor can wake up the node it is hosted on and can push this content to replicate it in the caches of other nodes that are currently in active phase. Nodes in active phase cache new content in their content store, the details depend on caching and replacement strategies. When the uplink turns up, all active nodes transmit their cached data to the uplink. We focus on scenarios where sensors monitor a phenomenon whereby **(i)** data relevance strictly decreases with time, and **(ii)** a more complete view of what the sensors are monitoring is achieved if available data comes from a larger number of distinct sources (i.e. sensors).

7.2.1.1 Uncoordinated Sleeping, Random Caching

In completely uncoordinated environments we cannot assume an administrative authority which pre-configures nodes. Each node thus decides every xD seconds whether it will be active or sleeping for the next xD seconds. The probability for active mode is given by the parameter p_a. Consequently, $1 - p_a$ describes the probability for a node to be awake. Active nodes that receive a content chunk will try to cache it with probability p_c, similar to the caching approach depicted in [Hail et al., 2015]. Sleeping nodes rely on the fact that p_a is chosen in a way that

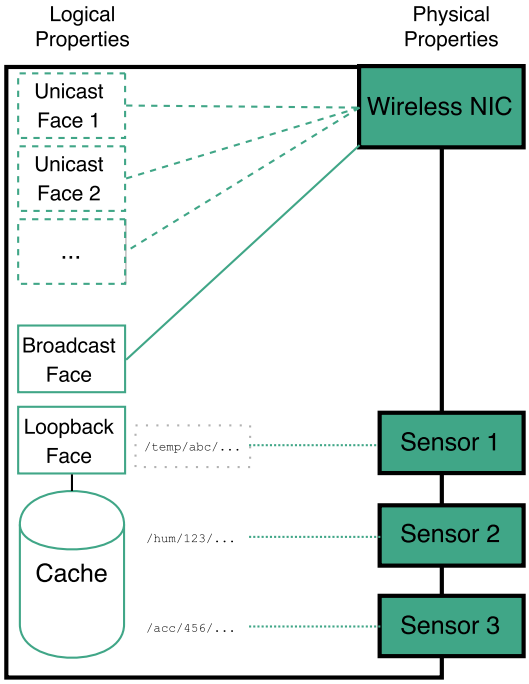


Figure 7.2: Architecture of an IoT device using NDN.

the expectation for active nodes is ≥ 1 at any given point of time to receive and cache their data on their behalf.

7.2.1.2 Coordinated Sleeping: Deputy on Watch (DoW)

Single Deputy This approach leverages coordination of nodes' sleeping phases. During the network's bootstrap, nodes determine an absolute order between them. The node in the first position of this order is elected as the first *deputy* and stays active for a certain period. Based on the determined order, each node will successively become *deputy* following a round-robin scheme. When a node wakes up to become the next deputy, it takes over deputy role by requesting the full cache from the previous deputy (using a simple Interest-based mechanism used by the uplink to request all available content from active nodes)[1]. Hence, with this approach only one node is on deputy watch at any time.

Multiple Deputies In scenarios where the amount of relevant content exceeds a single cache, a single deputy is not sufficient. Hence, we introduce multiple deputies responsible for different prefixes like */hum*. This can be pre-configured by the network operator.

[1]Details are described in Section 7.2.3.

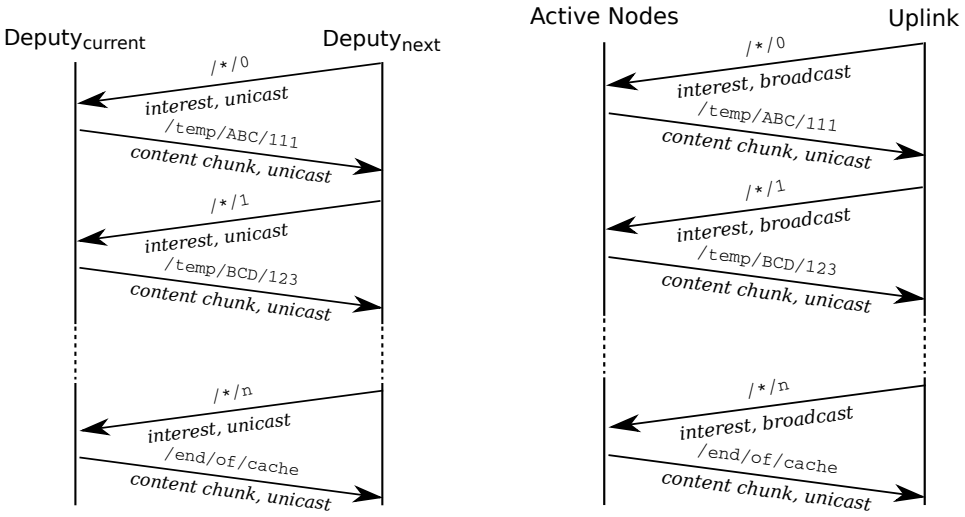
7.2.2 Name-based Caching Strategies

The caching and cache replacement strategy for the CS determines the content availability in terms of *diversity* and *freshness*. The caching strategy used as a baseline caches each chunk with a certain probability p_c. The basic cache replacement strategy is a First In, First Out (FIFO) policy, roughly equivalent to Least Recently Used (LRU) in this context. Some enhancements to these basic strategies are described in the following.

Max Diversity Most Recent (MDMR) Each node tries to cache each chunk ($p_c = 1.0$) and uses a different cache replacement strategy which favors diversity. The cache replacement strategy leverages the name of content and implements MDMR. In particular, the producer of the content can be identified by the combination of sensor type and node identifier. To implement a cache that maximizes diversity of content with respect to of different sensor sources, the IoT may benefit from the naming scheme in NDN. The age of the data can be determined from the timestamp component of the name. New content name is derived locally using the type of sensor (identified by the prefix of the name) and the timestamp (identified by the suffix of the name) see details in section 6.2.2. The cache replacement strategy works then as follows:

- First, the cache tries to replace older chunks from the same producer.
- Next, the cache tries to replace the oldest chunk of a producer from which several chunks are present in the cache.
- Again, the oldest value for the respective source will be replaced.
- Finally, if there is only entry per source, the oldest entry in the cache is replaced.

Prioritized Prefixes (P-MDMR) This approach works similarly to MDMR mechanism, but the caching and replacement strategy prioritize certain name prefixes. This prefix can be autoconfigured using local information coming from the (main) sensor of the node. A node with a temperature sensor, for example, will prioritize content for prefix */temp*. A node always tries to cache content chunks for the prioritized prefix, while other content are cached with a probability $p_c < 1.0$. Fig. 7.1 depicts a network with three different types of sensors and consequently three different prioritized prefixes scattered in the local IoT network. If the cache is full, entries for non-prioritized prefixes are replaced first.



(a) Distributing data between two caches.

(b) Distributing data from cache to uplink.

Figure 7.3: Basic communication schemes among deputies and between deputies and uplink.

7.2.3 Basic Implementation Requirements

In order to implement the mechanisms described above, little changes are necessary based on NDN:

Opportunistic Caching of Unsolicited Content When producers wake up, they want to offload content and immediately go back to sleep in order to save energy. The most basic approach is the distribution of new content via broadcast. In consequence, active nodes need to accept such broadcasts by configuring according *permanent PIT entries* as described in Section 6.2.3.

Interest Signaling for Group of Content When a new deputy wakes up, or when an uplink appears, cached content should be transferred. However, nodes may not be aware of previously distributed content and therefore cannot request each content chunk explicitly. To request data for an unknown name, we require a wildcard symbol (e.g., `<prefix>/*`), which expresses interest for all content under the prefix.

Having those mechanisms in place, nodes can locate and transfer content without relying on strictly synchronized schemes or significant prior knowledge. We summarize the traffic exchanges between producers and active nodes in Fig. 6.1, and between active nodes and uplink in Figure 7.3.

Symbol	Definition
$\|S\|$	Number of sensor sources
n	Number of sensor nodes
n_i	Number of designated caching nodes for content i
L	Lifetime of data
p_a	Probability of being awake
$1 - p_a$	Sleeping probability
p_c	Caching probability
p	Success probability $p_a \cdot p_c$

Table 7.1: Symbols and their definitions.

7.3 Evaluation

In order to examine the described strategies and evaluate energy consumption and content availability, a theoretical analysis as well as testbed and emulation based experiments were conducted.

7.3.1 Theoretic Model

For the theoretical analysis of the sleeping and caching strategies, we present basic models of cache replication among nodes, along with an efficiency estimate for a corresponding data collector. We consider uncoordinated sleeping, with a simple caching and replacement strategy.

7.3.1.1 Random Cache Selection

First we consider n equal nodes that act as data source and simultaneously provide caching capacity. As every node can hold several sensors, there are $|S| \geq n$ data sources, each producing data of a uniform lifetime L. Whenever a new sensor value is observed, a source node awakes, caches this data in its local cache, and broadcasts it to all listening neighbor nodes. We assume that L exceeds the content refresh period, and cache values are replaced, whenever new data from a content source arrives or the lifetime of some data is exceeded.

Neighbors are likely to sleep, but are awake with a common probability p_a. Once observed, broadcast data is included in receiver caches with finite probability p_c which needs not be 1 due to limited cache sizes. In our model, nodes are uncoordinated and while neglecting radio interference, we can assume independence of nodes and caches. Hence, data replication initiated by sources can be modeled as a Bernoulli experiment with success probability $p = p_a \cdot p_c$ at each replicator.

Consider R_i the effective number of replicas for content item i, then R_i follows a binomial distribution with

$$\mathbb{P}[R_i = r] = \left(\begin{array}{c} n-1 \\ r \end{array} \right) p^r (1-p)^{n-1-r}. \tag{7.1}$$

Hence, on average each content item is stored at the source and in caches throughout the network, i.e.

$$\mathbb{E}[\textit{content multiplicity}] = 1 + \mathbb{E}[R_i] = 1 + (n-1)p. \tag{7.2}$$

Content in our scenario has a (possibly significant) lifetime L and sensors may re-broadcast data in turns. Consider $R_i(L)$ the effective number of replicas for content item i experiencing L broadcast rounds. Then we can derive its distribution from Eq. (7.1) by including L 'losses' to successful trials, i.e.,

$$\mathbb{P}[R_i(L) = r] = \left(\begin{array}{c} n-1 \\ r \end{array} \right) p^r (1-p)^{(n-1)+(L-2)r}. \tag{7.3}$$

Whenever content is requested from the uplink, our sensor network carries data replicated according to Eq. (7.1), but nodes are likely to sleep. A content item is only available, when at least one caching node is awake, i.e., with probability

$$\begin{aligned} &\mathbb{P}[\textit{content availability}] \\ &= 1 - \sum_{r=0}^{n-1} (1-p_a)^{(r+1)} \, \mathbb{P}[R_i(L) = r] \\ &= 1 - (1-p_a) \left(1 - p \, + p\,(1-p_a)(1-p)^{L-1}\right)^{n-1} \end{aligned} \tag{7.4}$$

Data collectors on the upstream are interested in the ensemble of all $|S|$ content items at the same time. The expected outcome of collecting sensor data from the IoT network (modulo radio transmission errors) consists in all out of $|S|$ content items available at collection time. Hence

$$\begin{aligned} &\mathbb{E}[\textit{collectable content items}] \\ &= |S| \, (1 - ((1-p_a) \cdot \\ &\qquad \left(1 - p \, + p\,(1-p_a)(1-p)^{L-1}\right)^{n-1}\Big)\Big) \end{aligned} \tag{7.5}$$

7.3.1.2 Hardwired Cache Selection

Next we assume a "hardwired-content" model, where a fixed number n_i of designated caching nodes is selected for each content source i. In detail, each awake node

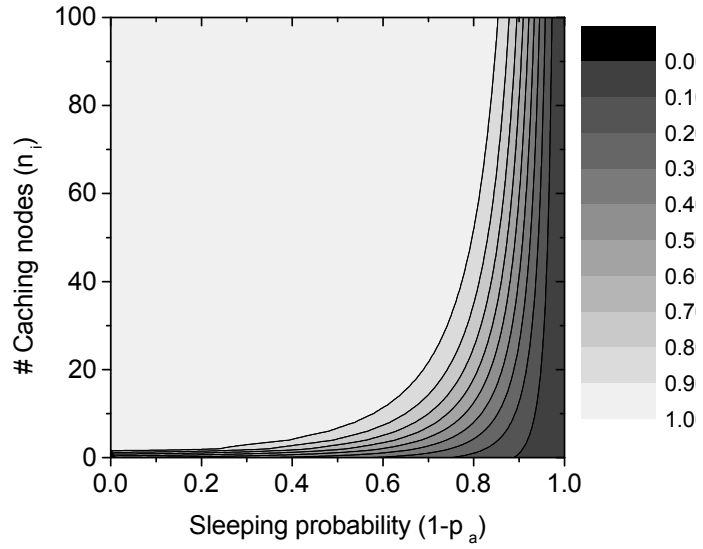


Figure 7.4: Content availability for $L = 1$. Hardwired cache selection model.

caches its dedicated content, whenever it is broadcasted. The problem then decomposes into node groups of sizes n_i, each of which only dependent on the probability p_a that a node is awake.

Correspondingly, Eq. (7.4) can be rewritten for this special case to

$$\begin{aligned} &\mathbb{P}[\textit{hardwired content availability}] \\ &= \quad 1 - \Big((1-p_a) \left(1 - p_a + p_a (1-p_a)^L\right)^{n_i - 1} \Big) \end{aligned} \tag{7.6}$$

Figure 7.4 shows predicted content availability for $L = 1$. Similarly, Eq. (7.5) transforms into

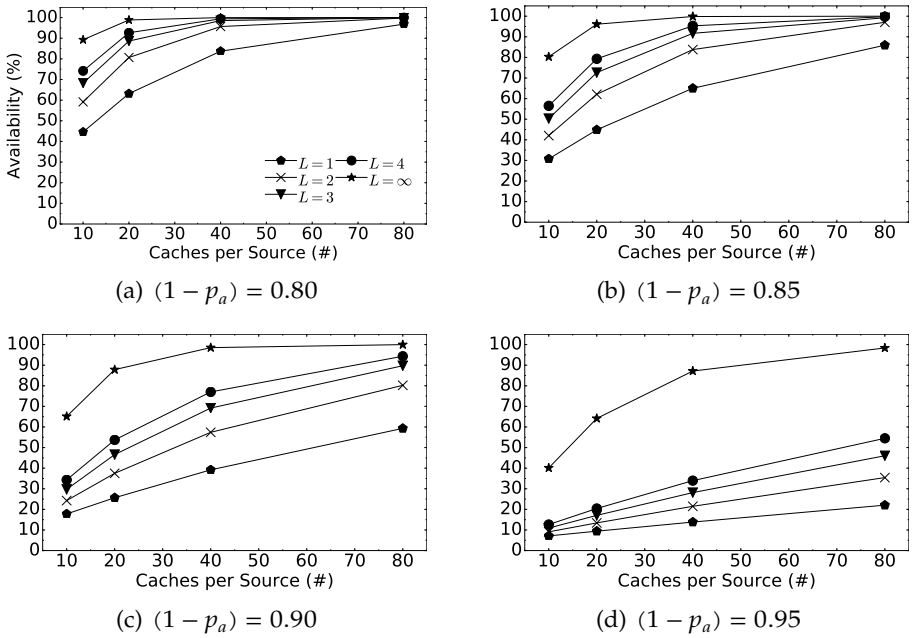
$$\begin{aligned} &\mathbb{E}[\textit{collectable hardwired content}] \\ &= \quad \sum_{i=1}^{|S|} \left(1 - \left((1-p_a)(1 - p_a + p_a (1-p_a)^L)^{n_i - 1}\right)\right) \end{aligned} \tag{7.7}$$

which simplifies to

$$\begin{aligned} &\mathbb{E}[\textit{collectable hardwired content}] \\ &= \quad |S| \left(1 - \left((1-p_a)(1 - p_a + p_a (1-p_a)^L)^{n_i - 1}\right)\right) \end{aligned} \tag{7.8}$$

in case all n_i are equal.

We will find model and experiments in excellent agreement in the subsequent section.



(a) $(1 - p_a) = 0.80$ (b) $(1 - p_a) = 0.85$

(c) $(1 - p_a) = 0.90$ (d) $(1 - p_a) = 0.95$

Figure 7.5: Theoretical model results (Eq. (7.8)): Availability as a function of n_i for various values of $1 - p_a$.

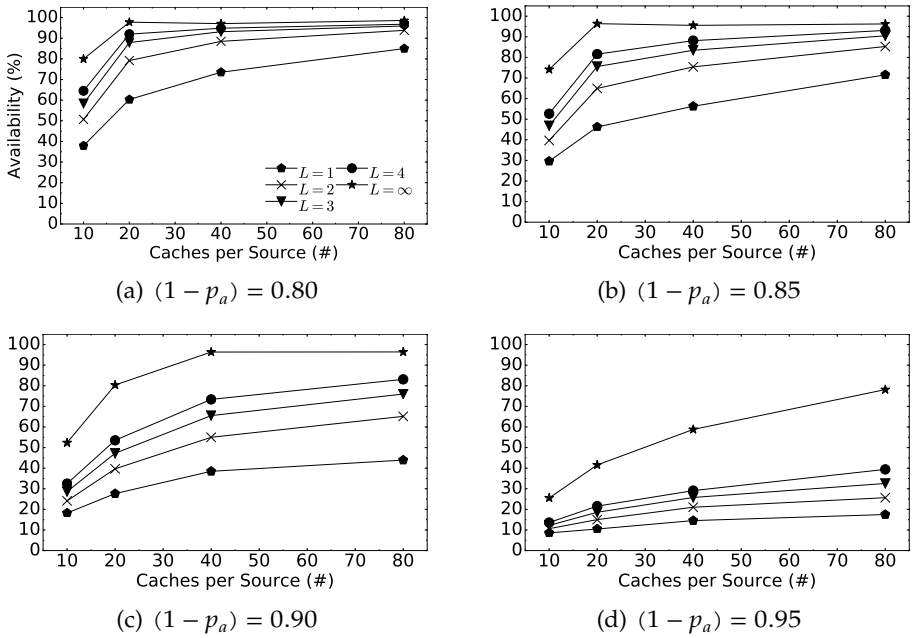
7.3.2 Experimental Evaluation

Implementation with RIOT and CCN-lite We implemented the caching, replacement, and sleeping strategies on top of RIOT. As presented in Section 4.2.5, RIOT supports the NDN implementation of CCN-lite as a *package*. We used some hooks in the CCN-lite protocol engine to implement small modifications to the processing of Interests and content chunks. For instance, a mechanism similar to the `mod_rewrite` module on HTTP daemons was introduced to rewrite Interests for `/*/`N on active nodes to match the Nth entry in the CS[2].

Experimental Setup on FIT IoT-LAB The experiments were conducted on up to 240 nodes nodes deployed over a 225 m^2, which are part of the Lille site of the FIT IoT-LAB testbed. Furthermore, we validated the results using an emulation tool: the *native* port of RIOT using a bridged virtual Ethernet connection with up to 1000 nodes.

Each run lasted for 30 minutes. Sources produced data in an interval between 1 and 35 s and the sleep/active cycle was set to 1 s, i.e. a node stays sleeping/active for a minimum timespan of one second. The uplink requested the content every 60 s. Nodes chosen for the experiments were of the IoT-LAB-M3 type.

[2]The chunks in the CS of each node have an arbitrary, but fixed order.



(a) $(1-p_a) = 0.80$ (b) $(1-p_a) = 0.85$

(c) $(1-p_a) = 0.90$ (d) $(1-p_a) = 0.95$

Figure 7.6: Experiment results: Availability as a function of n_i for various values of $1-p_a$.

In order to evaluate the energy consumption for these experiments, we measured the duration that a node spent in active and sleeping state plus the number of unicast and broadcast transmissions. The actual energy consumption of a deployment depends on the used energy model and the hardware. We chose a common energy model as proposed in [Schmidt et al., 2007]: $E = \sum_{state} P_{state} \cdot t_{state}$, where P_{state} defines the power consumed for a given *state* and t_{state} is the time to spent in this state. We define the states: *sleeping*, *active* (listening and receiving), *sending unicast*, and *sending broadcast* packets. Values for power consumption per state are taken from the datasheets of the MCU and radio transceiver. Furthermore, we assume a typical RDC rate of 0.6 %, i.e. the default value for ContikiMAC.

Experiment Results First, we validate the "hardwired-content" model as described in Section 7.3.1.2 via experimental results. As a derivation from our model, we vary two parameters: (i) the sleeping probability $1-p_a$ and (ii) the number of caching nodes per source. In practice, the upper limit for this second parameter is given by the memory constraint of the node.

We evaluate the availability of content items with respect to different lifetimes (L). We see that for $(1-p_a) \leq 0.85$, it is possible to obtain a good availability, even with a rather short lifetime of the content items, if enough caches are hardwired

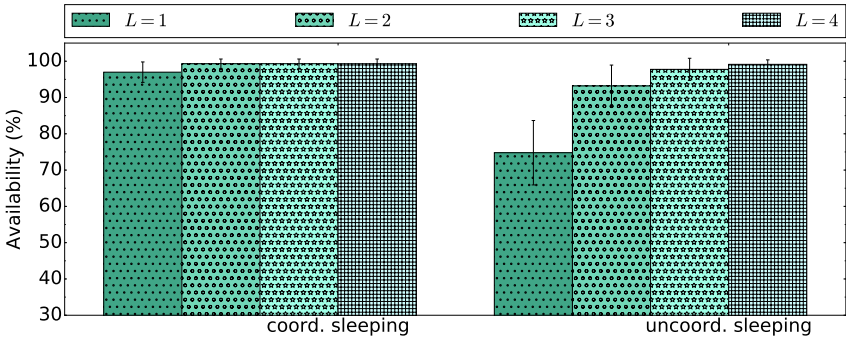


Figure 7.7: Comparing availability for different values of L between *coord. sleeping* and *uncoord. sleeping* approaches. The interval for a deputy was set to 30 s in the *coord. sleeping* approach and $(1 - p_a) = 0.8$ for the *uncoord.* one.

to the individual content items. Comparing the results from the experiments in Figure 7.6 to the values derived from the model as depicted in Figure 7.5, we see that both show similar trends. However, we see that the availability in the testbed results is slightly below the results from the model, particular for a high sleeping probability.

Next, we compare the results of the *uncoord. sleeping* with the performance of the *coord. sleeping*. Figure 7.7 reveals that for $(1 - p_a) = 0.8$ the *uncoord. sleeping* approach can achieve a similar high availability compared to the *coord. sleeping* one if we consider $L \geq 3$. However, for smaller values for L the *coord. sleeping* approach consistently outperforms *uncoord. sleeping*—unless p_a is drastically increased.

In order to evaluate the energy consumption for these experiments, we measured the duration each node spends in active and sleeping state, and the number of unicast and broadcast transmissions. We then fed these values to the energy consumption model presented in Section 5.4.1. We compare energy consumption of the "hardwired" scenario to two extreme cases: (i) a baseline where each node caches only the data it produces, has its CPU always on, but also uses state-of-the art radio duty-cycling with ContikiMAC and (ii) the *coord. sleeping* approach, where only one node (the *deputy*) is awake at any time. (Note that, hence, for the baseline, there is no communication between the nodes, only with the uplink). The results for the IoT-LAB-M3 nodes are shown in Figure 7.8. With *coord. sleeping*, more network traffic is induced, but nodes can spend longer time in sleep mode, which compensates the energy consumption of the communication overhead. We observe that we can reduce the energy consumption on a modern IoT platform by about 90 % compared to the baseline without affecting the data availability (compare Figure 7.7). A reason for that tremendous saving is that modern MCUs consume comparably much energy when active, while being very efficient when sleep-

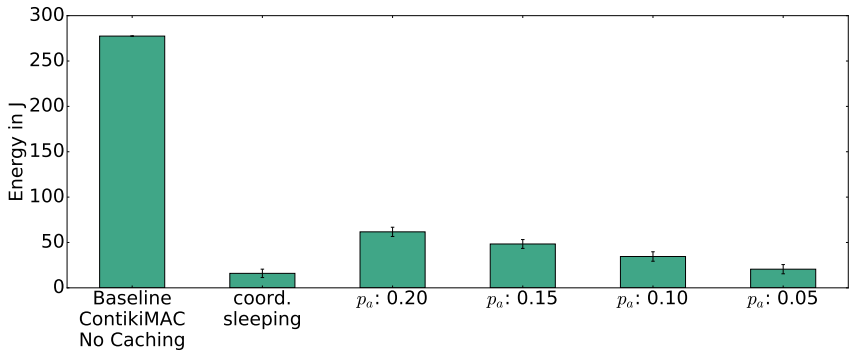


Figure 7.8: Modern IoT platform: Average energy consumption per node for *coord.* and *uncoord. sleeping* approaches on an ARM Cortex-M3 based node. ContikiMAC is used for radio duty cycling. Active MCU consumes 70 mA, listening consumes 12.8 mA and a broadcasts costs approx. 1.43 mJ.

ing. Besides, sending has become cheaper energy-wise with modern transceivers. Hence, the proposed approach where a node does not only perform radio duty cycling, but can also power down the CPU for most of the time, is much more energy efficient—even taking the additional traffic into account. In order to demonstrate the strong effect of the chosen hardware, we compare to a legacy WSN device. For that we apply the energy consumption data of a popular MSP430 based platform (TI MSP430161x MCU with cc1100 transceiver) to the model in Figure 7.9. On the one hand, we see that radio duty cycling (without additional CPU sleeping) has a much bigger impact. On the other hand, we observe a much stronger effect of the additional costs of packet sending, making the *coord. sleeping* approach less favorable.

7.4 Further Enhancement Strategies

In this section we describe additional caching strategies that go beyond the basic models as described in the previous sections. There are two main motivations for these additional strategies: **(i)** some of the assumptions and configurations from the previous sections are impractical for real deployments and **(ii)** performance with respect to the availability/sleeping ratio can be improved.

7.4.1 Replication Strategies

In the baseline replication strategy new content is only broadcasted once by its source. A small number of active nodes receive and cache this single broadcasted content—and interferences may further decrease the number of replicas of the con-

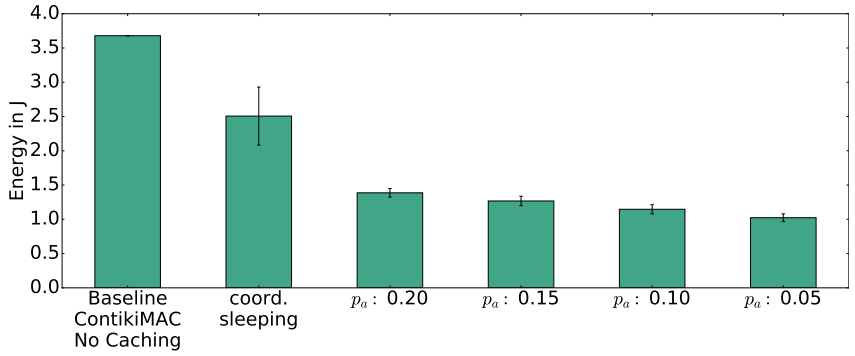


Figure 7.9: Legacy WSN platform: Average energy consumption per node for *coord.* and *uncoord. sleeping* approaches on an MSP430 based node. ContikiMAC is used for radio duty cycling. Active MCU consumes 0.5 mA, listening consumes 19.9 mA and a broadcasts costs approx. 2.8 mJ.

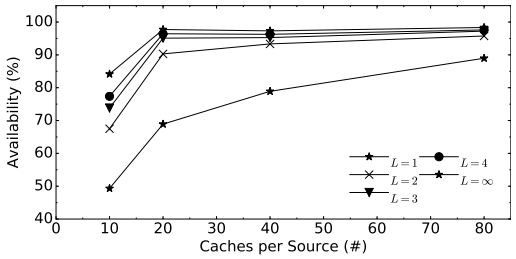


Figure 7.10: Availability as a function of n_i with source based replication.

tent. Experimental results confirms that data availability improves if one tolerates slightly less fresh data. This suggests that the freshest data may need to be replicated more, in order to improve available content freshness. We distinguish between two types of replication strategies **(i)** source-based replication and **(ii)** cache-based replication.

Source-Based Replication Strategy One problem that affects the content availability is caused by the fact, that a source broadcasts its content only once. Hence, in case of packet loss which may be caused by potentially bursty, external interference, a content chunk is definitely lost. A simple way to mitigating this risk can be implemented by additional replication of the data directly at the source. With this strategy, a source periodically rebroadcasts its latest content. In detail, a source that has just produced and broadcasted new content item sets a timer to be woken up again after some time t at which point it rebroadcasts this content (somewhat aged already, but still its freshest). t is ideally chosen in a way so that the timer

fires before the next sensor value is produced. This procedure could be performed multiple times with a smaller t.

In experiments on the testbed, it could be observed that replicating the latest content item only once already improves availability significantly. The results depicted in Figure 7.10 were gathered from an experiment with the same settings as the one depicted in Figure 7.6*b* ($(1 - p_a) = 80$) and reveal an improvement of availability by up to 20 %.

Cache-Based Replication Strategies
With this strategy, a node's active phase is split in two parts: **(i)** The *Renewal Phase*: the first part, which spans from wake-up to full cache renewal, i.e. the moment where each entry in the node's cache has been updated at least once since wake-up. **(ii)** The subsequent *Seeding Phase*, is the second part, which spans from full cache renewal until the node decides to go to sleep.

Pull-based Early Replication (PER) Nodes in the *Renewal Phase* have a partly outdated cache, which needs to be update as soon as possible. To speed up this process, nodes currently at the beginning of the *Renewal Phase* send a few Interests for their hardwired prefix ($/*$ by default), to which nodes that are currently in the *Seeding Phase* answer with random elements in their cache.

Push-based Seeding Replication (PSR) Nodes in the *Seeding Phase* have a completely up-to-date cache. In order to share the rarest part of their cache, nodes in the *Seeding Phase* periodically push the oldest entries in their cache (*middle-aged* content) to other currently active node's caches, via local broadcast.

These replication strategies take effect only if new content is produced at a sufficient rate, such that some nodes reach the *Seeding Phase* during their activity period. However, we have not evaluated these approaches and leave them for future work.

7.4.2 Autoconfiguration Mechanisms

7.4.2.1 Name Prefix Autoconfiguration

Using the approaches described in Section 7.3.1, each cache entry is *hardwired* to a particular source identified by its name. In practice, this has two drawbacks: **(i)** each cache needs to be preconfigured with a large number names, or needs to somehow gather this information during bootstrap, and **(ii)** a node needs to perform full name matching per received content chunk. The former is tedious at small scale and impossible at large scale. The latter constitutes a significant penalty for

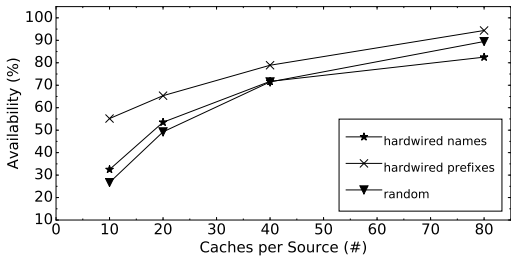


Figure 7.11: Availability as a function of n_i for hardwired names, hardwired prefixes, and randomized caching. $(1 - p_a) = 0.90$ and $L = 4$.

IoT devices that are equipped with a limited CPU and require to compute as little as possible in order to save energy.

One approach to deal with these problems could be to use a randomized caching strategy as proposed in [Hail et al., 2015], whereby each new content chunk received is cached with equal probability, and cache replacement is glsfifo. The experimental results presented in Figure 7.11 show this randomized approach performs similarly to the *hardwired* approach, and can even yield a better availability if the number of caches is large.

Another approach is to leverage potential knowledge about name structure. Thus, with this *hardwired prefixes* approach, a node can autoconfigure itself to cache content that have a name prefix matching its preferred sensor type (computationally much less expensive than matching the full name).

In Figure 7.11 we compare content availability on the FIT IoT-LAB testbed, in a similar setting as in the previous section, looking only at the case for $(1 - p_a) = 0.9$ and $L = 4$. The sources are equipped with three different types of sensors, i.e. they can be grouped in three different prefix classes. We observe that the hardwired prefixes approach achieves even better availability than the hardwired names approach. Next, we conducted experiments on RIOT *native* emulating 1,000 nodes (deploying five different sensor types). We first compared the results for hardwired names in a network with 240 nodes in the testbed to emulated network with 1,000 nodes using the same setup. From Figure 7.12 we observe that the numbers derived from the model in Section 7.3.1 do also hold for much bigger networks. Then we compared the hardwired names approach with the hardwired prefixes approach for the same 1,000 nodes network. In Figure 7.13 we see that availability significantly improves.

7.4.2.2 Dynamic Sleep Calibration

In this section, we describe autoconfiguration mechanisms for the sleeping strategy. In the long run, one can expect the number of nodes in the network may

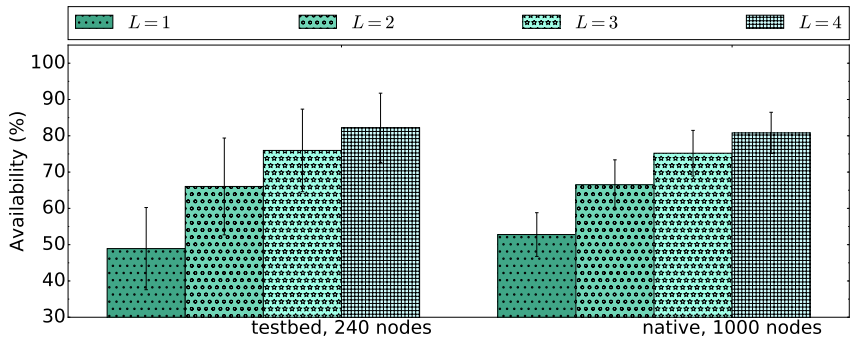


Figure 7.12: RIOT Emulator with 1,000 nodes: Comparing a network with 240 nodes in the testbed to the *native* emulator with 1,000 nodes for various values for L. $(1 - p_a) = 0.90$.

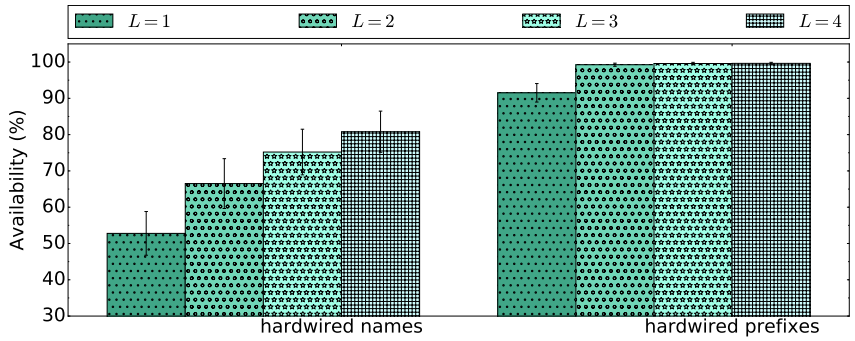


Figure 7.13: RIOT Emulator with 1,000 nodes: Availability as a function of n_i for hardwired names and hardwired prefixes. $(1 - p_a) = 0.90$.

vary. For instance, a substantial fraction of nodes temporarily runs out of energy (energy harvesting scenarios), or a batch of new nodes join the network. In such cases, dynamic calibration of sleep parameters may be necessary for the network to converge towards a state where IoT content availability can be sustained.

Temporary Keep-alive Based on passive monitoring of other nodes' activity (beaconing, e.g. PER-based signaling) nodes that are about to go to sleep can gauge by the rate of Interests they overhear for their hardwired prefix whether or not enough active nodes are currently available to cache content matching this prefix. In case this rate is below a given threshold, a node can decide to temporarily postpone going to sleep, in the hope that, in the mean time, more nodes will wake up to take over caching content matching this prefix.

Auto-adjusted Sleep Probability Based on passive monitoring of other nodes' activity (beaconing, e.g. PER-based signaling) nodes can estimate the number of sources and caches in the network, and based on Eq. 7.5 can derive the desired value p^* for sleep probability p, assuming that the tolerate freshness L is preconfigured. Locally on the node, an AIMD-like mechanism can be used to make the sleep probability p converge dynamically towards p^*. Periodically, each node adjusts p locally: if $p < p^*$, a node adds a small quantity to p, else, p is divided by fixed $f > 1$. We have not evaluated these approaches and leave them for future work.

7.5 Summary and Contributions

This chapter has shown how ICN mechanisms, most notably its caching abilities, can help to improve the energy efficiency by allowing nodes to sleep for a long time without harming the content availability. The results of the analysis, based on a theoretical model and testbed driven experiments, revealed that NDN's fundamentals with very few additional mechanisms can significantly improve the situation. One of these auxiliary mechanisms is MDMR, a new name-based caching strategy which has been designed, implemented, and studied.

Furthermore it presented novel mechanisms beyond MDMR, further improving reliability and/or energy efficiency. A new mechanism called Deputy on Watch (DoW) has been designed, implemented, and analyzed using coordinated sleeping to achieve a high degree of content availability, while requiring only a small number of nodes to be active and requiring only a minimum of coordination overhead. The analysis has shown that additional replication and autoconfiguration mechanisms can also improve the situation, when no coordination is desired or feasible.

Contributions
I have carried out extensive experiments with this implementation, both on real hardware with hundreds of IoT devices, and on an emulator with up to 1,000 emulated IoT devices. Testbed experiments I have conducted showed that the presented theoretical model is close to reality, providing a rather tight estimation of IoT content availability. I have also shown that the info-centric, cooperative caching mechanisms can achieve an order of magnitude reduction in energy consumption, while maintaining tolerably recent content availability above 90 %.

The work in this chapter was accepted for publication in the Proceedings of *IEEE Globecom Workshop Named Data Networking for Challenged Communication Environments (NDN-CCE)* [13].

Chapter 8

ICN over TSCH

Work presented in this chapter spawns from [13] and [15], which I co-authored with Emmanuel Baccelli, Cedric Adjih. Thomas C. Schmidt, and Matthias Wählisch.

In the previous chapter, we learned that by leveraging the native caching abilities of ICN, we can improve significantly the energy efficiency without harming content availability. However, two main challenges of typical IoT use cases, as presented in Section 1.2, are still open: **(i)** dealing with *packet loss* on the link layer and **(ii)** limiting the *latency*. While caching can help to improve content availability with sleepy nodes, it depends on a decent degree of PDR. In our experiments in Section 7.3.2, we observed that two simple mechanisms helped to improve the PDR:

1. Using the *Interest-Interest* approach as described in Section 6.2.3 increases the PDR by leveraging the link layer acknowledgements and retransmissions of IEEE 802.15.4 unicast traffic. However, this approach does not scale for too many active nodes with high density.

2. Simply retransmitting the broadcasting of unsolicited content in a preventive manner did also increase the PDR significantly. However, this approach does suffer heavily from bursty packet loss, e.g. caused by jamming. Moreover, broadcasting is often an energy-wise expensive operation if RDC or reservation-based MAC mechanisms are used and blindly retransmitting a broadcast is therefore not desirable.

Prior Work

Previous work has indicated that TDMA combined with channel hopping (i.e. TSCH) can significantly increase connectivity, efficiency, and stability of a network [Watteyne et al., 2009, Doherty et al., 2007, Duquennoy et al., 2015], achieving up to 99.99 % end-to-end reliability. Scheduling algorithms targeting TSCH were proposed as TASA [Palattella et al., 2012] and DeTAS [Accettura et al., 2013]. Tinka et al. proposed a simple scheduling mechanism for the TSCH MAC protocol that aims for full connectivity with a focus on mobile nodes and a dynamically changing neighborhood [Tinka et al., 2010].

8.1 The Idea of ICN over TSCH

A reservation-based MAC protocol, such as TSCH, increases robustness and determinism, but relies on upper layer services in order to compute and deploy an efficient transmission schedule (cf. Section 2.4.4). Hence, this section will discuss, how ICN can help to facilitate these tasks.

The need of an autoconfiguration mechanism for a transmission schedule in TSCH is twofold:

1. During the bootstrapping phase the mechanism needs to setup a schedule that assures local connectivity. The schedule must assure that nodes can communicate with their neighbors. It must also make sure that schedules of nodes in radio range do not conflict and collisions are avoided. A routing protocol may provide the scheduling algorithm with information about links to which neighbors are required to span the routing tree.
2. During normal operation of the network, the mechanism should adapt to the current traffic load between neighboring nodes and updates the schedule accordingly. This adaption is mandatory to achieve low end-to-end latencies.

In order to decrease the end-to-end latency of multihop communication, the schedule is ideally flow-ordered. To the best of our knowledge no autoconfiguration mechanism for setting up a TSCH schedule exists that adapts dynamically to changing traffic loads without prior or external knowledge about the traffic loads per node.

On the one hand, using ICN has several advantages:

- The symmetric nature of communication in ICN (Interest-Content pattern) can be leveraged to adapt dynamically to the network requirements and make estimations about link usage without a priori knowledge about the use case.
- ICN decreases the end-to-end latency by in-network caching which may mitigate the increased latencies in TSCH networks with low traffic load.
- Caching also reduces the average path length and therefore reduce the overall number of reserved cells in the network.
- Nodes can use Interest packets to request a new (or updated) schedule from a central entity or other nodes in the network without any prior configuration
- Scheduling information can be cached by ICN mechanisms inside the network and thus limit the traffic load needed for schedule distribution

On the other hand, LLNs pose challenges to ICN additional to the ones described in Section 6.2: Configuring sensible values for Interest timeouts and the number of Interest retransmission is difficult. Bursty packet loss due to external interference, multiple retransmissions on the link layer, or higher hop count may

require high values for timeouts and a higher number of Interest retransmissions. Thus, configuring the timeout for Interest retransmission too aggressive, may cause an unnecessarily high traffic load in the network, increasing the energy consumption and congestion. Then again, a high timeout value has a bad impact on the latency. Using a deterministic, collision-free, and interference-resilient link layer like TSCH, allows for a better estimation of timeout values, drastically decreases packet-loss and hence reduce the required number of retransmissions. Moreover, an increased reliability at the link layer supports the RPF. Moreover, RPF requires reliability at link layer which can be provided by TSCH, crucial for a proper deployment of ICN.

8.2 The Potentials for Link-Layer Adaptation

NDN Traffic Patterns Content distribution in NDN follows a request/response pattern with footprint on each hop. A request is propagated hop-by-hop in an Interest packet and implements a Pending Interest (PI) state in the corresponding tables (PIT) of intermediate nodes. Such a PIT entry matches at most one data chunk of limited size. Hence, in a fully deterministic, lossless setting, each request is answered by a train of up to k data packets within a time frame bound by the (temporal) diameter of the network.

For scheduling in TSCH, we can interpret an Interest as a predictor of data expected on the reverse path, and conversely can exclude any data arrival in the absence of PI state. We can further exploit the predefined chunk size for fixing the ratio of data per Interest packet in our schedule. Ideally, the arrival of an Interest would trigger the allocation of k slot frames towards the appropriate neighbor at the expected time.

However, as explained in Section 2.4.4, the dynamic reservation of cells requires coordination among neighbors and cannot be efficiently implemented chunk-wise. The *hidden terminal* problem in wireless networks requires a five-way handshaking scheme for negotiating the reservation [Zhu and Corson, 2001]. Consequently, this reservation procedure becomes an expensive task.

NDN Faces NDN introduces the concept of faces as an abstraction of logical network interfaces as discussed in Section 6.2.1. The use of a transmission schedule in TSCH allows to establish a cell-to-face mapping, where each cell (except for broadcast) is assigned to allow (unidirectional) transmission between individual nodes, only. Consequently, all scheduled cells within the transmission matrix of a node can be mapped to the corresponding faces.

Each face (except for a broadcast face) will typically consist of at least two cells—one RX (receive) cell and one TX (transmit) cell.

Channel hopping in TSCH enables data transmission within multiple cells at the same time. Spreading channels among faces will allow to schedule several faces in parallel. Thus, several adjacent links can be scheduled for the same timeslot without causing interference.

Design Aspects and Requirements

In our following design, we focus on a typical IoT deployment scenario of a wireless multi-hop network that can reach the Internet via at least one gateway. While the nodes may be constrained, the gateway is assumed to have sufficient memory resources for holding a full FIB. Furthermore, we assume a fairly static topology with mostly stationary nodes, since mobility is not in the focus of IEEE 802.15.4e [Palattella et al., 2013a].

8.3 Information-centric Networking Reservation Mechanisms

8.3.1 Schedule Construction and Maintenance

We now describe the design of a schedule for TSCH that is compliant to the ICN traffic pattern and adaptive to data demands. This shall flexibly optimize network performance and minimize energy consumption, but must not increase complexity for node coordination.

The general idea is a schedule that is partly static and pre-reserved, and partly dynamic and adaptive to the current traffic pattern. For this, we divide the slotframe into three parts, henceforth called subslotframes (*SSF*s). The first *SSF* is dedicated to statically scheduled Interest propagation and named SSF_I. Second, SSF_C is for sending back content chunks on a semi-dynamic schedule. The schedule of the third *SSF* is fully dynamic. This SSF_{Dyn} is activated to serve increased traffic loads on dedicated links.

For the following description of the scheduling procedure, we define $G = (V, E)$ as an undirected graph with a set of vertices V representing the set of nodes and a set of edges E representing the links between two nodes present in the routing graph. If two nodes a and b share an edge $(a, b) \in E$, they are called *1-hop neighbors*.

SSF_I – Static Interest Schedule The cells in this first subslotframe are reserved at network bootstrapping after the topology is created (or reconfigured). For reconfiguration purposes, the reservation of the first cell ($c(1, 1)$) is fixed to a general broad-

cast (of entire wireless range) and used to alert all nodes within wireless reach. Nodes that do not need to send any reconfiguration data, are required to switch to receiving mode for slot 1 at channel offset 1. Each node reserves a predefined number of TX cells to each of its 1-hop neighbors, and a matching RX cell (same slot number, same channel offset) for each TX cell a 1-hop neighbor has allocated towards it. In this way, basic capacities for exchanging Interests among neighbors are defined. The amount of reserved cells per neighbor can be chosen according to a priori knowledge of communication patterns—upstream (or default) routes may receive higher capacities, for example.

Additionally, a node should reserve cells for broadcasting to cope with incomplete routing information. Broadcast capacities may be aligned with predictable traffic patterns and available FIB memory. Interest broadcasts are limited to 1-hop neighbors and different from the general broadcast in cell $c(1,1)$.

$\mathbf{SSF_C}$ – Semi-dynamic Content Schedule Each Interest is potentially answered by a content chunk. Taking this information into account and assuming a maximal chunk size of k packets, the content schedule in the second *SSF* shall be built as follows. For each RX cell in SSF_I, a node reserves k TX cells, and for each TX cell in SSF_I, a node reserves k RX cells. As such, the cell assignment does *not* require any negotiations between nodes, but is a direct consequence of the SSF_I, and static.

However, the nature of NDN traffic allows for an adaptive operation of the SSF_C. Initially, all reserved cells are deactivated, which means that the transceiver will not be switched on and the CPU may remain in energy saving mode. Node b activates k RX cells for a neighboring node a, after an Interest has been sent to a in SSF_I. These cells will get deactivated again, either after a content chunk was received from a, or when the PIT entry times out and is removed. By deactivating cells, energy can be saved from reducing idle listening and increasing the time the CPU can spend in sleep mode.

In the case of Interest broadcasting, these savings cannot apply. To limit broadcast reception periods, we assign shared cells to SSF_C. A TSCH shared cell operates CSMA/CA for increased flexibility at the price of reduced reliability.

$\mathbf{SSF_{Dyn}}$ – Dynamic On-Demand Schedule Cells in the third part of the slotframe stay unreserved at bootstrapping, and are only activated if traffic demands exceed the initially foreseen capacities. On a per link base, a balanced set of Interest and content cells are (de)allocated dynamically between two nodes and adapt the wireless spectrum to current utilization patterns. In detail, each node monitors the utilization of the (directional) links to each of its neighbors. Link utilization U is measured as the ratio between *used* cells c_u and *scheduled* cells c_s: $U = c_u/c_s$. A

cell is called *used*, if a node i send a packet and a neighboring node j receives this packet in c. A cell is called *scheduled*, if it is reserved as a TX cell in i's schedule and reserved as an RX cell in j's schedule.

If the recent link utilization U_{cur} from node a to node b over a pre-defined time period T exceeds a predefined threshold U_{Th}, a and b reserve a preconfigured set of additional slots for sending/receiving Interests and content in SSF_{Dyn}. Thresholds and allocated slot sizes are parameters of the network that can be adjusted to meet deployment-specific criteria (see example below). Deallocation is performed after the U_{cur} falls below a certain threshold U_{Tl} in T. In this way, radio resources can be dynamically adapted to actual (bursty) traffic demands that may vary between node pairs, while low (regular) communication requirements allow for extended sleeping cycles in radio interfaces and thus enhance energy efficiency.

The dynamic adaptation of the schedule requires coordination between 1-hop and 2-hop neighbors. The information about a node schedule and the schedule of its 1-hop neighbors can be piggy-backed in ICN (Interest) traffic in a memory-efficient representation (such as bit fields). In this manner, a node will gain knowledge about the schedules of all nodes within its 1-hop and 2-hop neighborhood. This information serves as basis for reserving additional cells in SSF_{Dyn} by a link scheduling protocol like *LAMA*.

Example Assuming a typical building automation scenario nodes may request periodic configuration and software updates—e.g., provided by gateway acting as the root node (1) in the routing tree. Taking this knowledge into account, nodes will make more reservations in SSF_I for upstream packets. Let a slotframe consist of 101 slots (as proposed by the IETF 6TiSCH WG) and 16 channel offsets (according to the 16 channels available in IEEE 802.15.4). For simplicity we assume furthermore that $k = 1$. A sensible partitioning could be to assign 20 slots to SSF_I and SSF_C respectively. Depending on the network's density a node may reserve 1 (high density) to 9 (very low density) cells per neighbor in each of the first two SSFs. The remaining 60 slots—remember that the first slot is reserved for broadcasting—are assigned to SSF_{Dyn} and thus unreserved in the beginning. While the cells reserved in SSF_I and SSF_C may suffice the general requirements for fetching and delivering configuration information, it may happen from time to time that more data has to be delivered to the downstream nodes, e.g. in case of a firmware update. In this case, nodes will detect a high utilization of the cells in SSF_I and SSF_C and according make reservations for these links in SSF_{Dyn}. Hence, up to 30 additional cells may be reserved for Interests and content chunks respectively. After the firmware update is fully delivered to the affected nodes, reservations in SSF_{Dyn} can be deallocated again.

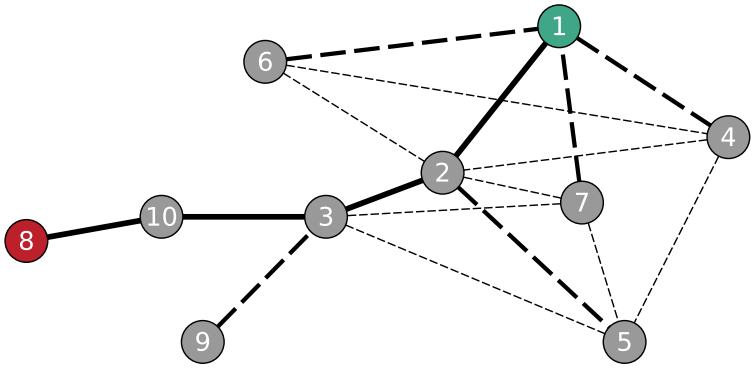


Figure 8.1: Testbed topology in the experiments. A connection between two nodes indicates that this link has been scheduled in TSCH. Thick lines are part of the formed DODAG.

It can be seen that the sizes of ideally SSF_I and SSF_C should be kept considerably small and only ensure basic connectivity, in order to assign more cells to SSF_{Dyn}.

8.3.2 Evaluation

Experiment Setup

In order to evaluate our scheduling solution, we conducted experiments in the FIT IoT-LAB. We compare the approach with an implementation that runs ICN directly on the link layer, using CSMA as a MAC protocol as discussed in the previous chapters. As a hardware platform we used the IoT-LAB-M3 node. The software is based on the de-facto standard implementation of IEEE 802.15.4e, OpenWSN [OpenWSN, 2016] on top of RIOT.

Ten nodes are chosen, forming a multi-hop topology shown in Figure 8.1. Node 8 acts as the consumer and node 1 as the content provider as well as the root node in the routing tree. The requested content consists of 100 chunks. We assume side traffic from nodes connecting to a (sub)networks. Therefore, nodes 4, 6, and 7 are also generating traffic with a similar rate as the content consumer (node 8).

MAC Configurations

The static schedule and the routing tree were computed beforehand. The static schedule for SSF_I and SSF_C ensures basic connectivity and reserves one cell per link and direction. We use a length of 15 ms for the slot length and 101 slots per slotframe. The remaining cells in the slotframe are left initially unscheduled and can be reserved in SSF_{Dyn}. The schedule is constructed in a way that packets can

travel from any node along the tree to the root node and from the root node to any node within one slotframe in SSF_I and SSF_C respectively. Time synchronization between the nodes is done based on periodic broadcasting of enhanced beacons in shared cells.

For the first series of experiments all scheduled cells in SSF_I and SSF_C were active and node 8 was sending out Interests with a constant rate of one Interest per slotframe. We refer to this configuration as *Static Information-centric Networking Reservation (SINR)*.

In the second series of experiments the scheduled cells in SSF_C were kept initially inactive. Again, node 8 was sending out Interests with a constant rate of one Interest per slotframe. As soon as a node A on the path from 8 to 1 receives an Interest, it activates its RX cell(s) in SSF_C on the link to the next hop B on the path. Once, A receives the corresponding content chunk from B it deactivates the cell again. We refer to this configuration as *Dynamic Information-centric Networking Reservation (DINR)*.

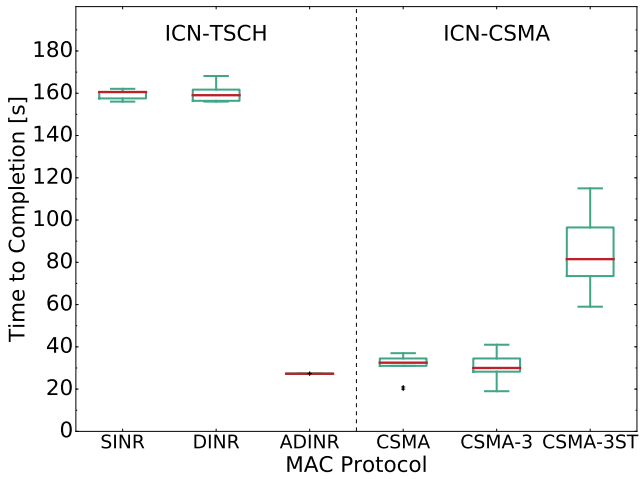
The next series had the same configuration as for *DINR*, but made also use of the dynamic part of the schedule SSF_{Dyn}. If a node A receives a certain amount of Interests from one of its neighbors, they implicitly activate cells in SSF_{Dyn} in both directions to increase the bandwidth on this link. If the cells for this link are less frequently used, the additional cells are deactivated again. In this configuration we increased the rate in which node 8 generates to 15 Interests per slotframe. We refer to this configuration as *Adaptive Dynamic Information-centric Networking Reservation (ADINR)*).

We compared the results for *SINR*, *DINR*, and *ADINR* with different configurations of ICN on top of a CSMA MAC protocol. In all configurations, a node initiates up to three link layer retransmissions if no ACK is received. Interests are retransmitted after a timeout of 1 second by node 8 if no content chunk has been received. In the first configuration, simply referred to as *CSMA*, node 8 retransmits Interests until it has received the whole content. The second configuration, referred to as *CSMA-3*, limits the number of Interest retransmissions to three tries. The last configuration, referred to as *CSMA-3ST*, is similar to *CSMA-3*, but with increased traffic from nodes 4, 6, and 7.

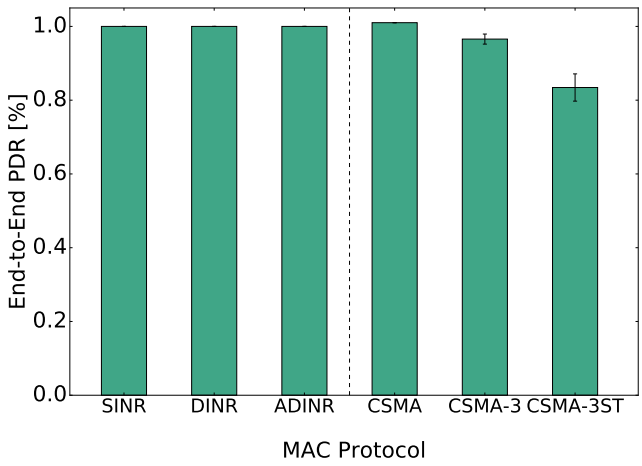
Each serie of experiments is sampled with the same parameter settings until it is converged.

Results

We considered four different metrics: **(i)** time to completion, **(ii)** jitter, **(iii)** end-to-end PDR, and **(iv)** energy consumption.



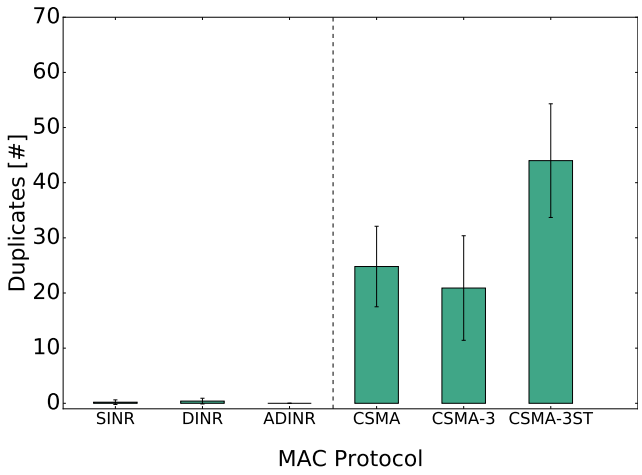
(a) Time to Completion



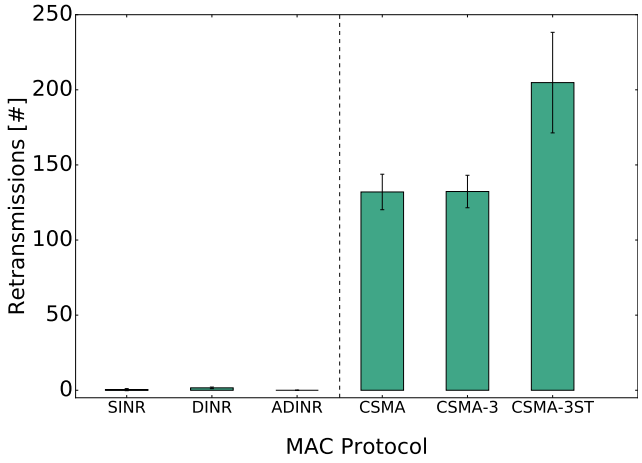
(b) Packet Delivery Ratio

Figure 8.2: Comparison of time to completion and PDR in different configurations for TSCH and CSMA.

Since only one Interest and one content chunk can be transmitted per slotframe with *SINR* and *DINR*, the minimum time to completion for fetching 100 content chunks is $\Delta = 100 * T_{SF}$ with T_{SF} being the duration of the slotframe. As we can see in Subfigure 8.2(a), the measured time is only slightly above this minimum. Initially, *ADINR* generates more Interests per slotframe than it can send out, but gradually, nodes along the path activate more cells in SSF_{Dyn}. SSF_{Dyn} contains 70 cells which implies that up to 8 additional links per hop (4 hops, bidirectional) can be scheduled. This leads to a tremendous improvement of the time to completion in comparison to *SINR* and *DINR*. Concerning jitter, our measurements in



(a) Duplicates



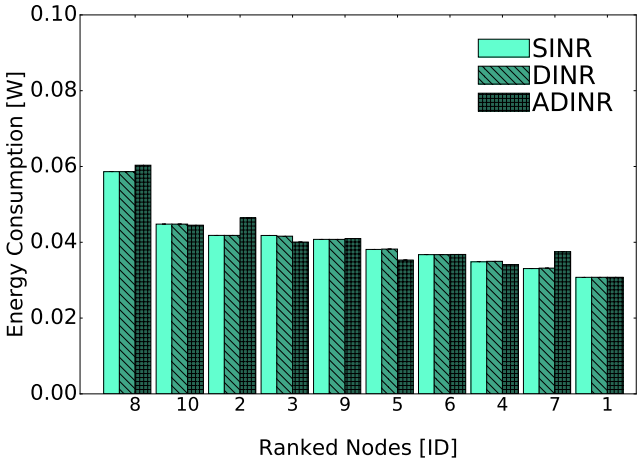
(b) Retransmissions

Figure 8.3: Number of duplicates and retransmissions in different configurations for TSCH and CSMA.

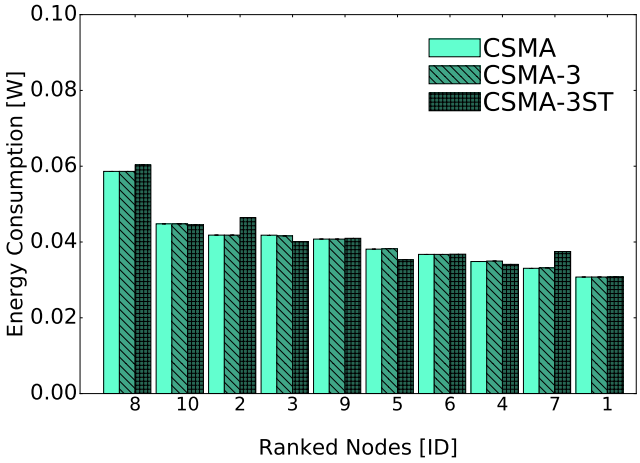
Subfigure 8.2(a) show a very small standard deviation for SINR, DINR, and ADINR, as expected for a reservation-based MAC.

In comparison, time to completion with CSMA is much less predictable and depends heavily on the number of collisions and retransmissions (per link and end-to-end). We observe significantly bigger standard deviation and increasing average if side traffic increases

As expected with a collision-free TSCH schedule, we observed almost no link layer retransmissions with *SINR*, *DINR*, and *ADINR* (less than 5 retransmissions overall). Consequently these mechanisms achieved an end-to-end PDR of 100 % for all three TSCH configurations, as shown in Subfigure 8.2(b). With *CSMA*, Inter-



(a) TSCH



(b) CSMA

Figure 8.4: Energy consumption for the different configurations of TSCH and CSMA.

ests are retransmitted as many times as required, and thus end-to-end PDR reaches 100 % too, but at the cost of many retransmissions and duplicates as shown in Figure 8.3. On average, we counted more than 130 end-to-end retransmissions and 25 duplicate chunks that arrived at the consumer. Limiting the number of end-to-end retransmissions to three (in *CSMA-3*, see Subfigure 8.2(b)) decreases the PDR to about 97 %, with similar numbers for retransmissions and duplicates. If side traffic increases (in *CSMA-3ST*) the PDR drops further down, with significantly more retransmissions and duplicates.

The energy measurements were performed using the control nodes provided by FIT IoT-LAB (power consumption measurement through resistor shunts and an

INA226 current/power monitor component). We configured the INA226 with a conversion time of 8244 ms and the averaging mode to 1024 which gives maximum accuracy according to the hardware datasheet. We computed the average over all samples in all experiment runs, per node, as shown in Figure 8.4. With TSCH, transceivers switch to sleep mode for all unscheduled slots. Thus, we can see that all nodes consume consistently less power with *SINR*, *DINR* and *ADINR* than with *CSMA*. Furthermore, we observe that the increased energy consumption in *ADINR* due to a higher duty cycle is leveled out by the fact that the nodes can more quickly return to sleep mode again.

8.4 Summary and Contributions

This chapter has studied how the TSCH MAC layer can be leveraged in ICN scenarios. On the one hand, it can be observed how TSCH increases the reliability and determinism of the network and can thus help to configure ICN in a more sensible way. On the other hand, it can be concluded that ICN's symmetric traffic pattern can be leveraged to compute a link-layer transmission schedule for TSCH networks. As a consequence, a novel Information-centric reservation mechanism has been proposed and evaluated in three configurations: SINR, DINR, and ADINR. Using experiments in the testbed, I have studied how these approaches can be leveraged to increase the reliability compared to CSMA-based deployments and decrease the energy consumption at the same time.

Contributions

While the reliability of TSCH avoids the need for complex error recovery mechanisms at the ICN layer, TSCH combined with appropriate dynamic scheduling allows to be more energy efficient, while not incurring more delays, compared to typical contention-based MAC layers. I have thus designed mechanisms adjusting jointly, and on-the-fly, both TSCH time slot reservations and ICN Interest/Chunk multi-hop routing. I have shown the efficiency of these mechanisms through experiments on state-of-the-art IoT hardware. Compared to ICN on common contention-based MAC layers, I have observed that my approaches **(i)** offer similar delay and throughput performance, but **(ii)** are more reliable and more predictable, and **(iii)** (iii) can reduce energy consumption by almost 50 %.

The work in this chapter was published in the Proceedings of *ACM conference on Information-centric Networking (ICN)* [29].

Chapter 9

Conclusion

In order to enable Smart Object networks at Internet-scale and thus, fulfill the vision of the IoT, the key concepts of today's Internet need to be adopted. These key concepts can be categorized into technical and non-technical aspects. The fundamental technical aspects are [Feldmann, 2007]: **(i)** a layered design, **(ii)** a packet switching paradigm, **(iii)** network of collaborating networks, and **(iv)** the end-to-end principle. The main non-technical aspects are: **(i)** interoperable open standards, **(ii)** deployment of open source software[1], and **(iii)** no centralized governance.

Protocols based on open standards like the ones specified by the IETF, have proven to constitute a solid foundation for today's Internet and provide the required flexibility and interoperability, in order to implement the IoT. While popular Internet protocols, like TCP or HTTP, are not applicable for the IoT one-to-one, concepts like end-to-end acknowledgements or REST can be adopted. As a consequence, the IoT requires a combination of existing protocol specifications and algorithms, modified to match particular IoT requirements, along with new protocols and algorithms.

In contrast to this approach on open standards, proprietary silo solutions are similar to WSN solutions, where various systems operate isolated from each other and can only communicate through special gateways. The design of these application specific silo approaches may lead to the best performance or best efficiency for one particular use case, but hinders flexibility. Since it cannot be foreseen which application domains will arise in future IoT scenarios, the above mentioned Internet concepts allow to adapt to many different requirements and interoperable open standards guarantee the versatility of this approach.

In order to tackle the particular challenges of IoT scenarios, other standards than only IP-based ones need to be considered, too. ICN is one of the most promising alternative networking approaches, providing a number of advantegous characteristics. Leveraging ICN's in-network caching abilities can significantly reduce traffic demands and reduce the energy consumption while maintaining a high content

[1] Currently, about 70 % of the top million busiest web sites are running Apache or nginx open source web server software [Netcraft, 2016], more than 80 % of visible Internet mail servers run Exim or Postfix open source mail server software [Security Space, 2015], and more than 96 % of the top one million servers run Linux as an OS [W3Cook, 2016].

availability. Due to its simple architecture it can reduce the complexity for IoT networks and thus, require less memory. Since ICN is transport agnostic, it can be easily deployed directly using link layer technologies, making it very efficient for LLN scenarios, or over IP, allowing it to be deployed over today's Internet architecture. This thesis demonstrates how information-centric approaches can be successfully deployed for IoT use cases, reducing the energy consumption without harming the other requirements.

On the software side Linux has indisputable been a success story on a global scale, similar to IP on the protocol side. In fact, similar as IP is the narrow waist in the Internet protocol stack, Linux can be seen as the narrow waist of the Internet software stack. The key for Linux' success is based on its open source, license and community model. This model, for instance, allows researchers to implement and evaluate new approaches on Linux and apply the resulting implementations directly in practice. Moreover, this model permits commercial users to run proprietary software in conjunction with an open source software environment. As a consequence, Linux was massively adopted for business use cases which lead to significant support from many companies over the time [Tecmint, 2014]. This thesis presents RIOT as a software platform to operate low-end IoT devices. RIOT aims to play a similar role for the IoT as Linux does for the traditional Internet, adopting Linux' model, but providing an architecture and implementation that addresses the particular requirements of constrained devices and LLNs.

Wireless multi-hop communication is a common scenario for these LLNs in the IoT. Research in this domain demands for experiment-driven research based on testbeds [Blywis et al., 2010a]. Additionally, to a solid and versatile software platform (e.g., RIOT), which is necessary to implement network protocols and algorithms on IoT devices, this approach also requires specialized tools that enable a deeper understanding of the system. For this purpose, traditional software engineering tools, such as debuggers or profilers, have to be extended and modified to reveal the necessary insights. An emulator—emulating both, hardware and the wireless topology—is an important facility to enable many of these tools. This thesis describes some of these tools for experiment-driven research on IoT scenarios.

9.1 Perspectives

ICN Standardization ICN reveals some promising characteristics for IoT scenarios. However, in order to enable ICN for Smart Object networks at Internet-Scale, more standardization efforts are required. The Internet Research Task Force (IRTF) has already started to work on collecting the prerequisites for specification, but no WG has been chartered inside the IETF or any other relevant standardization body

so far. Standardization must not only consolidate the various directions of ICN and come up with a unified protocol specification, but also tackle some of the most critical challenges for ICN in the IoT. Among the open issues are routing, naming schemes, and defining the role of the gateway.

Potentially asymmetric or volatile links in LLNs pose new challenges to the task of routing in ICN, particular for the RPF. Additional mechanisms to check the bi-directionality of links periodically or on demand are one way to tackle this issue, but increase the complexity and traffic load of the system. Gossip like routing approaches may also be considered in order to reduce the state.

Names inside a LLN do not necessarily need to be unique, but are required to be short. Name compression schemes might be a possible approach to achieve this, but may conflict with aggregation mechanisms. Mapping names to link-layer addresses, using multicast mechanisms, where available, is also advisable.

The transition between a wired backbone where ICN is transported on top of IP and LLNs where ICN can act directly on top of the link layer may happen seamlessly. However, a gateway between these types of networks still plays a particular role. It can translate between different namespaces or filter certain traffic to reduce the number of transmissions inside the LLN.

RIOT Development RIOT provides a versatile open source software platform to develop IoT solutions. It offers a comprehensible API, allows for well-known programming paradigms and easy integration of third-party code. However, for the anticipated ubiquity of IoT systems, it is inevitable that not only highly specialized embedded software engineers can develop IoT applications. American National Standards Institute (ANSI) C may offer an efficient and flexible foundation, but is ill-suited for fast and easy application development. Using stripped-down versions of scripting languages, e.g., MicroPython [MicroPython, 2016], eLua [eLua, 2016], or JerryScript [JerryScript, 2016], are one approach to tackle this challenge.

Following current trends in hardware design and the vision of *smart dust*, it can be envisioned that future IoT systems will be even more distributed then current deployments. Smaller, cheaper, and more energy efficient chips with a specialized feature set are supposed to work tightly together. For these systems, RIOT's multi-threaded architecture where most of the central APIs (e.g., netapi) are IPC-based, could provide a suitable basis.

There won't be a one-fits-all solution for the IoT—neither on the hardware nor on the software side. Hence, providing flexible and configurable systems is a basic requirement in the IoT. On the OS level this demands not only for an easy integration of frameworks and libraries, but also requires to provide configurability on the kernel level. Consequently, future versions of RIOT should allow the devel-

oper to choose between different scheduling strategies, programming paradigms, or memory models.

In the end, just like Linux which is 25 years after its initial release still under heavy development, the work on RIOT will probably also never be "done".

Security and Privacy Issues in the IoT The ubiquitous nature of IoT applications makes the deployment of efficient and effective security and privacy preserving mechanisms an imperative. The mere amount of IoT devices in combination with the lack of interfaces makes secure bootstrapping and key distribution a big challenge for the IoT. Consequently, new methods for key exchange are required. Leveraging proximity information or out-of-band signaling, short range radio technologies, such as NFC, or visual light communication are potential candidates for this key exchange.

Similar to information-centric networking, new approaches towards security in the IoT may focus rather on securing the content than on securing the communication channel. Such a security scheme offers several advantages in highly dynamic systems such as LLNs, where maintaining a stable, secure channel between two devices may be challenging and costly. Encrypting and signing content objects instead allows for mobility of the endpoints and facilitates scenarios where caching proxies or gateways are involved.

While interaction between humans and computers so far happened mostly via well-defined input devices like a keyboard, pointer device, or touch screen, the interaction in future IoT scenarios will be pervasive and en passant. On the one hand, this poses new challenges to security and privacy, on the other hand, the decentralization of this approach also provides new opportunities. Transitioning from applications where everything is connected to the *cloud*, to solutions that operate mostly via local ad-hoc communication and move the services closer to the user, makes it significantly harder to monitor or manipulate the communication.

It has become clear over the recent years, that governments and business corporations are putting tremendous efforts into pervasive surveillance. New trends like *big data* or *machine learning*, new technologies like *quantum computing*, and the enormous progress of *artificial intelligences* are severe threats to privacy in the 21st century. However, the new era of ubiquitous connectivity of literally everything may constitute a good countermeasure to this threat—if security is implemented correctly—due to the decentralization and vast amount of communication channels of this paradigm. Enforcing openness and transparency are further important steps to improve security and privacy levels.

Appendix A

Résumé Français

L'Internet des Objets a pour but d'intégrer des milliards d'objets connectés dans l'Internet. Du point de vue matériel, ces objets connectés sont de petits ordinateurs très bon marché, basés sur des micro-contrôleurs et des puces radio efficaces en énergie apparus récemment, couplés avec des capteurs et actionneur divers, le tout alimenté par une batterie de très petite taille. Ces objets connectés sont donc typiquement très contraints en ressources telles que CPU, mémoire et énergie. De plus, les liens radio à travers lesquels communiquent les objets connectés ont une capacité très limitée, sont souvent sujet à des taux de perte importants, et peuvent requérir du routage spontané entre objets connectés pour fournir la connectivité nécessaire. Ces caractéristiques posent des défis, d'une part en termes de logiciel embarqué s'exécutant sur les objets connectés, et d'autre part en termes de protocols réseaux utilisés par les objets connectés pour communiquer. En conséquence, de nouvelles méthodes et outils expérimentaux sont nécessaires pour étudier in vivo les réseaux formés d'objets connectés, de nouvelles plateformes logicielles sont nécessaires pour exploiter efficacement les objets connectés, et des protocoles de communication innovants sont nécessaire pour interconnecter ces objets. La présente thèse relève en partie ces défis, en introduisant des nouveaux outils facilitant l'utilisation de grands réseaux test interconnectant de nombreux objets connectés, un nouveau système d'exploitation (RIOT) utilisable sur une très grande variété d'objets connectés, ainsi que plusieurs nouveaux mécanismes utilisant le paradigme des réseaux centré contenus pour améliorer significativement l'efficacité énergétique des protocoles de communication standards de l'Internet des Objets. Les principales contributions de cette thèse sont résumées ci-dessous.

La première contribution consiste en une analyse des besoins pour la conception d'un système d'exploitation répondant aux exigences particulières de l'Internet des Objets, passant en revue les principales alternatives conceptuelles pour de tels systèmes d'exploitation, à la fois d'un point de vue technique et d'un point de vue non-technique, en se concentrant sur les systèmes d'exploitation open source et généralistes.

La deuxième contribution est RIOT, un nouveau système d'exploitation basé sur un micro-noyau qui répond aux exigences des cas d'utilisation de l' IoT. Cette thèse montre que RIOT est aussi frugal en ressources que Contiki et TinyOS, les

deux systèmes d'exploitation pionniers des plateformes logicielles pour l' IoT. De plus, RIOT offre un compromis avantageux entre fonctionnalité et performance du système d'exploitation, tout en permettant l'efficacité la portabilité du code IoT pour une large gamme de matériels et de cas d'utilisation. RIOT démontre que les aspects de programmation exotiques imposés par Contiki et TinyOS ne sont pas nécessaires sur le matériel IoT. Cette thèse présente une étude détaillée du micro-noyau RIOT et de l'abstraction matérielle, ainsi que du sous-système réseau. La thèse donne également un aperçu des aspects non-techniques concernant la communauté open source de RIOT.

La troisième contribution concerne l'étude de la consommation d'énergie dans l' IoT. J'ai développé DES-eProf, un nouvel outil pour mesurer la consommation d'énergie sur les systèmes IoT, à divers niveaux allant de l'analyse basée sur les fonctions ou des threads à l'évaluation des blocs fonctionnels. Contrairement aux approches similaires dans ce domaine, DES-eProf ne nécessite aucune modification des pilotes de périphérique ou du code d'application.

La quatrième contribution concerne l'efficacité énergétique des protocoles réseaux pour l' IoT. J'ai étudié la faisabilité, les avantages et les défis d'une approche basée sur les réseaux centrés-contenu (ICN) dans l' IoT. La thèse fourni les premières expériences d' ICN dans un déploiement à taille réelle de l'Internet des Objets, répartis sur plusieurs dizaines de pièces à plusieurs étages d'un bâtiment. L'utilisation de mécanismes ICN standard et des améliorations que j'ai proposées diminuent de manière significative le trafic de contrôle nécessaire au fonctionnement du réseau et exploite efficacement les caches pour répondre aux exigences IoT en termes d'énergie et de contraintes de bande passante. Cette évaluation fournit également la première comparaison expérimentale de l' ICN avec les normes IoT dominantes: 6LoWPAN, RPL et UDP.

Ensuite, j'ai analysé comment les stratégies de mise en cache ICN peuvent être exploitées pour améliorer la disponibilité de contenu dans les scénarios IoT avec mise en veille de nœuds. En effet, les nœuds IoT doivent être mis en veille le plus possible afin de réduire la consommation d'énergie. Dans le même temps, les données produites par ces appareils doivent être disponibles à tout moment. En l'absence d'un cache centralisé qui n'est lui, jamais en veille, un compromis est nécessaire entre la consommation d'énergie et la disponibilité des données IoT. J'ai donc proposé une nouvelle approche de mise en veille coordonnée, appelée Deputy on Watch, ainsi qu'une stratégie de mise en cache et de remplacement du cache nommée MDMR. Cette thèse montre que que les appareils peuvent ainsi réduire leur consommation d'énergie d'un ordre de grandeur tout en maintenant la disponibilité de contenu IoT supérieure à 90 %.

Enfin, j'ai étudié une approche améliorant la fiabilité et de déterminisme du réseau tout en maintenant une faible consommation d'énergie en combinant l' ICN et le Time-Synchronized Channel Hopping (TSCH), une technologie de couche de liaison sans fil de plus en plus populaire dans l' IoT. La thèse montre comment les modèles de communication ICN peuvent être exploités pour optimiser dynamiquement le calcul de l'ordonnancement pour TSCH, avec un nouveau mécanisme de réservation adapté à ICN. Je montre que, comparé à CSMA, cette nouvelle approche est plus robuste face aux interférences sans fil, tout en consommant moitié moins d'énergie, et fourni un meilleur compromis entre débit et délai.

List of Figures

List of Tables

Acronyms

6LoWPAN IPv6 over IEEE 802.15.4 networks. ix, xi, 3, 14, 15, 29–31, 43, 46, 48, 53, 70, 71, 76–78, 85, 90, 97, 99, 101–103, 108, 142, 147, 152–157, 194, 199, 201, 231

6TiSCH IPv6 over the TSCH mode of IEEE 802.15.4e. 30, 43, 44, 108, 182

6lbr 6LoWPAN Border Router. 31, 38, 48, 153, *Glossary:* 6LoWPAN Border Router

6ln 6LoWPAN Node. 48

6lr 6LoWPAN Router. 48

6top 6TiSCH Operation Sublayer. 30, 43

ADC analog digital converter. 129, 133, 134

ADINR Adaptive Dynamic Information-centric Networking Reservation. 184–186, 188

AES Advanced Encryption Standard. 39, 41

AMI Advanced Metering Infrastructure. 11

ANSI American National Standards Institute. 25, 87, 88, 109, 110, 191

AODV Ad-hoc On-demand Distance Vector routing. 32, 33, 48

API Application Programming Interface. 1, 12, 37, 38, 47, 57, 61, 64, 66, 67, 75, 76, 91, 92, 94–96, 100, 103–105, 107, 109, 110, 113, 191

ASIC Application-Specific Integrated Circuit. 129, 130

ASN absolute slot number. 44

ASN.1 Abstract Syntax Notation One. 41

BFS breadth-first search. 43

BLE Bluetooth Low Energy. 26, 28, 29, 46, 73

CBOR Concise Binary Object Representation. 41, 42, 202, 237

CCN Content-Centric Networking. 50, 147, 155, 156, 216

CI continuous integration. 67, 88

CoAP Constrained Application Protocol. 4, 35, 36, 38, 41–43, 46, 52, 53, 152, 153, 155, 156, 202, 205, 216, 231, 233, 236

CoCOA CoAP Simple Congestion Control/Advanced. 35

CoMI CoAP Management Interface. 42, 43

CoRE Constrained RESTful Environments. 36, 52

COSE CBOR Object Signing and Encryption. 41

CS Content Store. 50, 163, 168

CSMA Carrier Sense Multiple Access. xi, 14, 26–30, 49, 183–188, 195, 198

CTS Clear To Send. 34

DDS Data Distribution Service. 37, 38, 206

DECT Digital Enhanced Cordless Telecommunications. 29

detnet Deterministic Networking. 22

DINR Dynamic Information-centric Networking Reservation. 184–186, 188

DMA direct memory access. 134

DoW Deputy on Watch. 162, 176

DTLS Datagram Transport Layer Security. 40–42, 46

ECC Elliptic Curve Cryptography. 39, 40

ETSI European Telecommunications Standards Institute. 110, 111

ETX Expected Transmission Count. 31, 119

FH-CDMA Frequency-Hopping Code Division Multiple Access. 26–28, 43, 49

FIB Forwarding Information Base. 32, 143, 146, 147, 150

FIFO First In, First Out. 163

FSM Final State Machine. 128, 129

GCC GNU Compiler Collection. 65, 99, 131, 221

GDB GNU Debugger. 65, 88

GPIO General Purpose Input/Output. 94, 99

GPL GNU General Public License. 48, 68, 70–74

HC Header Compression. 29, 97

HTTP Hypertext Transfer Protocol. 1, 36, 42, 46, 97, 168, 189

HVAC Heating, Ventilation, and Air Conditioning. 8, 9, 147, 148, 150

ICMP Internet Control Message Protocol. 30, 32

ICN Information-Centric Networking. ix–xii, 14–16, 50, 51, 107, 108, 141–143, 145, 150, 152, 156, 157, 159, 176–180, 182–184, 186, 188–191, 194, 195, 199, 210–212, 226, 228, *Glossary:* Information-Centric Networking

IEEE Institute of Electrical and Electronics Engineers. 3, 11, 110

IETF Internet Engineering Task Force. ii, 3, 12, 19, 22, 29–32, 36, 41, 42, 47, 49, 55, 56, 67, 110, 111, 182, 189, 190, 209, 214–218, 220, 221, 223, 225–228, 230, 232–234, 236–238

IoT Internet of Things. ii, iii, vii–ix, xi, 1, 3, 4, 10–17, 19–30, 32, 34–44, 46, 48, 50–81, 83, 85–88, 90–94, 97–102, 104, 108–117, 121–127, 130, 131, 136, 137, 139, 141–148, 150, 152–157, 159–163, 166, 170, 171, 174–177, 180, 188–195, 197, 198, 211, 213, 215, 230, 231, 233

IP Internet Protocol. 4, 9, 15, 25, 29, 30, 40, 46, 47, 54, 55, 73, 86, 88, 101, 103, 104, 107–109, 145, 152, 153, 155, 189–191, 203

IPC Inter Process Communication. xii, 64, 66, 75, 76, 87, 89, 91, 92, 103–107, 191

IPsec Internet Protocol Security. 39, 40

IPSO IP for Smart Objects. 46, 47, 110

IPv4 Internet Protocol version 4. 1, 25, 71, 105, 108, 228

IPv6 Internet Protocol version 6. xi, 3, 4, 15, 19, 25, 29, 30, 32, 36, 39, 46, 54, 62, 68, 70, 71, 76, 88, 97, 99, 101, 103, 105, 108, 153, 197, 201, 213, 216, 219, 223, 228, 234, 238

IRTF Internet Research Task Force. ii, iii, 190, 228

ISM Industrial, Scientific and Medical. xi, 9, 49

ISR interrupt service routine. 93, 105, 132

ITU International Telecommunication Union. 46, 49

JSON JavaScript Object Notation. 42

JTAG Joint Test Action Group. 65, 123

KVM Kernel Virtual Machine. 125

L2CAP Logical Link Control and Adaptation Protocol. 49

LGPL GNU Lesser General Public License. 68, 86, 111

LLN Low-power and Lossy Network. vii, ix, 19–21, 24, 25, 27, 29–36, 38–43, 45–47, 51, 54, 90, 116, 120, 136, 142–144, 146, 148, 150, 152–154, 156, 157, 160, 178, 190–192

LNA Low Noise Amplifier. 127, 128

LRU Least Recently Used. 159, 163

LWM2M OMA Lightweight M2M. 42, 43, 47

MAC Medium Access Control. xi, 14, 16, 22, 25–30, 41, 43–45, 49, 51–53, 71, 78, 103, 105, 108, 142, 177, 178, 183, 184, 188, 224, 227, 232

MANET Mobile Ad-hoc NETwork. 11, 32, 33, 46, 141

MCU microcontroller. xii, 1, 3, 10, 11, 20, 21, 39, 52, 57, 60, 61, 63–65, 70–74, 76, 78, 88–95, 97, 101, 102, 123, 132, 133, 143, 160, 169–172

MDMR Max Diversity Most Recent. 14, 16, 163, 176, 194

MIB Management Information Base. 41, 42

MMU Memory Management Unit. 62, 64, 66, 73, 78, 90, 91

MOP Mode of Operation. 32, 33

MPU Memory Protection Unit. 91

MQTT MQ Telemetry Transport. 37, 38, 47, 204

MQTT-SN MQTT for Sensor Networks. 37

MTU Maximum Transmission Unit. 29, 34, 148, 153

mutex mutual exclusion. 66, 89, 109

ND Neighbor Discovery. 24, 29, 30, 101, 153

NDN Named-Data Networking. 14, 50, 51, 142–157, 159, 161–164, 168, 176, 179, 181, 197, 199, 211

NETCONF Network Configuration Protocol. 41, 234

NFC Near Field Communication. 29, 101, 192

NIC Network Interface Controller. 142, 143

NTP Network Time Protocol. 43

OF Objective Function. 88

OIC Open Interconnect Consortium. 110

OLSR Open Link State Routing. 32, 33

OMA Open Mobile Alliance. 3, 42, 110, 204

OMG Object Management Group. 3, 37, 38

OS operating system. 1, 2, 12, 13, 15, 34, 38, 46, 49, 50, 55–81, 85, 86, 88–90, 92, 94–100, 102, 104, 106, 108–110, 112–114, 116, 122, 128–131, 155, 189, 191, 197, 199

OSCoAP Object Security CoAP. 41

PA Power Amplifier. 127

PAN Personal Area Network. 48, 49

PCE Path Computation Element. 30

PDR Packet Delivery Ratio. 16, 125, 177, 184–187, 198

PDU Protocol Data Unit. 34

PIT Pending Interest Table. 50, 143–145, 150, 159, 164, 179, 181, *Glossary:* Pending Interest Table

PKI Public Key Infrastructure. 40

PLC Power Line Communication. 5, 50, 58

PM Power Mode. 88, 89

POSIX Portable Operating System Interface. 25, 67, 71, 96, 105, 109, 110

PSK Pre-Shared Key. 23, 40

PTP Precision Time Protocol. 43

QoE Quality of Experience. 31

QoS Quality of Services. 31, 36–38

RDC Radio Duty Cycling. 26, 28, 169, 177

REST Representational State Transfer. 36, 38, 43, 189

ROLL Routing over Low power and Lossy networks. 19, 31

RONR Reactive Optimistic Name-based Routing. 14, 15, 146, 147, 149–151, 155, 157, 197

RPF Reverse Path Forwarding. 50, 143, 179, 191

RPL Routing Protocol for Low-Power and Lossy Networks. xi, 4, 14, 30–33, 44, 70, 88, 98, 99, 102, 103, 142, 147, 155, 156, 194

RSA Rivest-Shamir-Adleman. 39

RSSI Received Signal Strength Indication. 31

RTO Retransmission Timeout. 34, 35

RTOS Real-Time Operating System. viii, 58, 70, 71, 73–80

RTS Ready To Send. 34

sDDS Sensor networks DDS. 38

SDP Service Discovery Protocol. 49

SECG Standards for Efficient Cryptography Group. 39

SIG Special Interest Group. 49

SINR Static Information-centric Networking Reservation. 184–186, 188

SME Small to Mid-sized Enterprise. 80, 111

SNMP Simple Network Management Protocol. 41, 119

SoC System on Chip. 76

SPI Serial Peripheral Interface. 65, 94, 99

TBMS Testbed Management System. 118, 125, 137

TCP Transmission Control Protocol. 34, 35, 37, 40, 41, 51, 73, 77, 97, 103–105, 108, 189, 209, 218, 221

TDMA Time Division Multiple Access. 27, 43, 44, 105, 177, 222

TLS Transport Layer Security. 40

TLV Type Length Value. 42

TSCH Time-Synchronized Channel Hopping. x, 14, 16, 27–30, 43–45, 51, 53, 54, 108, 177–188, 195, 197, 198, 201

UART Universal Asynchronous Receiver/Transmitter. 94, 97, 99, 108, 221

UDP User Datagram Protocol. xi, 14, 29, 33, 35–37, 40, 46, 51, 98–100, 102, 103, 105–108, 126, 147, 155, 194

UI User Interface. 63, 66

VANET Vehicular Ad-Hoc Network. 57

VIF Vanilla Interest Flooding. 146, 148–150

WBAN Wireless Body Area Network. 6, 7, 11

WG Working Group. 19, 22, 30–32, 36, 47, 111, 182, 190

WMN Wireless Mesh Network. 22, 26, 31, 32, 34, 43

WSAN Wireless Sensor and Actor Network. 112

WSN Wireless Sensor Network. vii, 1–4, 9, 11, 21, 24, 43, 55, 56, 58, 64, 70, 74, 75, 79, 85, 90, 107, 109, 112, 120, 127, 129, 130, 134, 141, 189

WWW World Wide Web. 36

Publications of the Author

[1] O. Hahm, M. Günes, and K. Schleiser, "DES-Testbed A Wireless Multi-Hop Network Testbed for future mobile networks," in *5th GI/ITG KuVS Workshop on Future Internet*. Stuttgart, Germany: GI/ITG KuVS, 2010.

[2] ——, "The DES-Framework - Extending a Wireless Multi-Hop Testbed by virtualization and simulation," in *10th Würzburg Workshop on IP: Joint ITG, ITC, and Euro-NF Workshop "Visions of Future Generation Networks" (EuroView2010)*. Würzburg, Germany: ITG/Euro-NF, 2010. [Online]. Available: http://www.euroview2010.com/data/abstracts/Session1_2_Hahm_Guenes_Multi-Hop_Testbed.pdf

[3] O. Hahm, S. Adler, N. Schmittberger, and M. Günes, "Poster Abstract: Energy Profiling for Wireless Sensor Networks," in *GI/ITG KuVS Fachgespräch Sensornetze*. GI/ITG KuVS, 2011.

[4] O. Hahm, M. Günes, F. Juraschek, B. Blywis, and N. Schmittberger, "An Experimental Facility for Wireless Multi-Hop Networks in Future Internet Scenarios," in *The 2011 IEEE International Conference on Internet of Things*, IEEE. Dalian, China: IEEE, 2011.

[5] O. Hahm, N. Schmittberger, and M. Günes, "Implementation of a TCP/6LoWPAN stack for Wireless Sensor Networks," 2012. [Online]. Available: http://www.kuvs-ngsdp.org/_slides/08_Implementation-TCP-6LoWPAN-stack_Hahm.pdf

[6] O. Hahm and S. Adler, "Profiling energy consumption of Wireless Sensor Nodes with almost zero effort," in *First International Workshop on Novel approaches to Energy Measurement and Evaluation in Wireless Networks*, IEEE. Ottawa: IEEE, 2012.

[7] O. Hahm, E. Baccelli, and K. Schleiser, "Painless Class 1 Devices Programming," IETF, 2013. [Online]. Available: http://tools.ietf.org/html/draft-hahm-lwig-painless-constrained-programming-00

[8] O. Hahm, E. Baccelli, M. Günes, M. Wählisch, and T. C. Schmidt, "Poster Abstract: OS for the IoT: Goals, Challenges, and Solutions," in *Proceedings of the French Interdisciplinary Workshop on Global Security (WISG)*, 2013, poster.

[9] O. Hahm, E. Baccelli, H. Petersen, M. Wählisch, and T. C. Schmidt, "Demonstration Abstract: Simply RIOT: Teaching and Experimental Research

in the Internet of Things," in *Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '14, IEEE. Piscataway, NJ, USA: IEEE Press, 2014, pp. 329–330. [Online]. Available: http://dl.acm.org/citation.cfm?id=2602339.2602399

[10] O. Hahm, S. Pfeiffer, and J. Schiller, "On Real-time Requirements in Constrained Wireless Networks for Mobile Health," in *Proceedings of the 4th ACM MobiHoc Workshop on Pervasive Wireless Healthcare*, ser. MobileHealth '14, ACM. ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2633651.2637475

[11] O. Hahm, C. Mehlis, L. Ortmann, T. Eichinger, and M. Lenders, "Betriebssysteme für eingebettete Systeme im Internet der Dinge: Freie Fahrt für Experimentierfreudige," *iX Developer Magazine, Special Issue on Embedded Software*, 2014.

[12] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: a Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, October 2016.

[13] O. Hahm, C. Adjih, E. Baccelli, T. C. Schmidt, and M. Wählisch, "Poster Abstract: ICN over TSCH: Potentials for Link-Layer Adaptation in the IoT," in *Proceedings of the 3rd ACM conference on Information-centric Networking (ICN-2016)*, ser. ICN '16, ACM. ACM, 2016.

[14] O. Hahm, E. Baccelli, M. Wählisch, T. C. Schmidt, and C. Adjih, "A Named Data Network Approach to Energy Efficiency in IoT," in *Proceedings of Global Communications Conference: Workshops: Named Data Networking for Challenged Communication Environments*, IEEE. IEEE, 2016.

[15] ——, "Time Slotted Channel Hopping and Information-Centric Networking for IoT," in *Proceedings of the 8th IFIP International Conference on New Technologies, Mobility and Security*, IFIP. IEEE, 2016.

[16] E. Baccelli, O. Hahm, M. Wählisch, M. Günes, and T. Schmidt, "RIOT: One OS to Rule Them All in the IoT," 2012. [Online]. Available: http://hal.inria.fr/hal-00768685

[17] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Poster Abstract: RIOT OS: Towards an OS for the Internet of Things," in *32nd IEEE INFOCOM*, IEEE. Turin, Italy: IEEE, 2013.

[18] E. Baccelli, C. Mehlis, O. Hahm, T. C. Schmidt, and M. Wählisch, "Information Centric Networking in the IoT: Experiments with NDN in the Wild," in *Proceedings of the 1st ACM conference on Information-centric Networking (ICN-2014)*, ser. ICN '14, ACM. ACM, 2014.

[19] E. Baccelli, G. Bartl, A. Danilkina, V. Ebner, F. Gendry, C. Guettier, O. Hahm, U. Kriegel, G. Hege, M. Palkow, H. Pertersen, T. Schmidt, A. Voisard, M. Wählisch, and H. Ziegler, "Poster Abstract: Area & Perimeter Surveillance in SAFEST using Sensors and the Internet of Things," in *Workshop Interdisciplinaire sur la Sécurité Globale (WISG2014)*, Troyes, France, 2014. [Online]. Available: https://hal.inria.fr/hal-00944907

[20] E. Baccelli, O. Hahm, H. Petersen, and K. Schleiser, "RIOT and the Evolution of IoT Operating Systems and Applications," *ERCIM News*, vol. 2015, no. 101, 2015. [Online]. Available: http://ercim-news.ercim.eu/en101/special/riot-and-the-evolution-of-iot-operating-systems-and-applications

[21] 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 *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)*, ACM. ACM, May 2015.

[22] H. Petersen, C. Adjih, O. Hahm, and E. Baccelli, "Demo Abstract: IoT Meets Robotics - First Steps, RIOT Car, and Perspectives," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16, ACM. USA: ACM, 2016, pp. 269–270. [Online]. Available: http://dl.acm.org/citation.cfm?id=2893711.2893767

[23] K. Schleiser, O. Hahm, and M. Günes, "Poster and Abstract: G-Mesh-Lab - A Wireless Multi-Hop Network Testbed for the G-Lab," in *TridentCom 2010*, EAI. EAI, 2010.

[24] B. Blywis, M. Günes, F. Juraschek, and O. Hahm, "Properties and Topology of the DES-Testbed," Tech. Rep. TR-B-11-02, 2011. [Online]. Available: http://edocs.fu-berlin.de/docs/receive/FUDOCS_document_000000009836

[25] B. Staehle, F. Wamser, S. Deschner, A. Blenk, D. Staehle, O. Hahm, N. Schmittberger, and M. Günes, "Application-Aware Self-Optimization of Wireless Mesh Networks with AquareYoum and DES-SERT," in *11th Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop "Visions of Future Generation Networks" (EuroView2011)*, Würzburg, Germany, 2011. [Online]. Available: http://www.euroview2011.com/fileadmin/content/euroview2011/abstracts/abstract_staehle.pdf

[26] F. Juraschek, M. Günes, M. Philipp, B. Blywis, and O. Hahm, "DES-Chan: A Framework for Distributed Channel Assignment in Wireless Mesh Networks," in *Proceedings of the Australasian Telecommunication Networks And Applications Conference (ATNAC 2011)*, 2011.

[27] T. Watteyne, V. Handziski, X. Vilajosana, S. Duquennoy, O. Hahm, E. Baccelli, and A. Wolisz, "Industrial Wireless IP-Based Cyber–Physical Systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1025–1038, 2016.

[28] C. Gündogan, C. Adjih, O. Hahm, and E. Baccelli, "Let Healthy Links Bloom: Scalable Link Checks in Low-power Wireless Networks for Smart Health," in *Proceedings of the 6th ACM International Workshop on Pervasive Wireless Healthcare*, ser. MobiHealth '16, ACM. ACM, 2016, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/2944921.2944924

[29] O. Hahm, C. Adjih, E. Baccelli, T. C. Schmidt, and M. Wählisch, "A Case for Time Slotted Channel Hopping for ICN in the IoT," *CoRR*, vol. abs/1602.08591, 2016. [Online]. Available: http://arxiv.org/abs/1602.08591

References

[6lo, 2016] 6lo (2016). IPv6 over Networks of Resource-constrained Nodes. https://datatracker.ietf.org/wg/6lo.

[Aberdour, 2007] Aberdour, M. (2007). Achieving quality in open-source software. *IEEE software*, 24(1):58–64. IEEE.

[Accettura et al., 2013] Accettura, N., Palattella, M. R., Boggia, G., Grieco, L. A., and Dohler, M. (2013). Decentralized traffic aware scheduling for multi-hop low power lossy networks in the internet of things. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th Int. Symposium and Workshops on a*, pages 1–6. IEEE.

[Adjih et al., 2015] Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., and Watteyne, T. (2015). FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT)*. IEEE.

[Ahlgren et al., 2012] Ahlgren, B., Dannewitz, C., Imbrenda, C., Kutscher, D., and Ohlman, B. (2012). A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36. IEEE.

[Akyildiz et al., 2002] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422. Elsevier.

[Akyildiz and Vuran, 2010] Akyildiz, I. F. and Vuran, M. C. (2010). *Wireless sensor networks*. John Wiley & Sons.

[AllSeen, 2016] AllSeen (2016). AllJoyn. https://allseenalliance.org/framework.

[Amadeo et al., 2014a] Amadeo, M., Campolo, C., and Molinaro, A. (2014a). Internet of Things via Named Data Networking: The support of push traffic. In *Network of the Future (NOF), 2014 International Conference and Workshop on the*, pages 1–5. IEEE.

[Amadeo et al., 2014b] Amadeo, M., Campolo, C., and Molinaro, A. (2014b). Multi-source data retrieval in IoT via named data networking. In *Proceedings of the 1st international conference on Information-centric networking*, pages 67–76. ACM Press.

[Amini et al., 2007] Amini, R., Gill, E., and Gaydadjiev, G. (2007). The challenges of intra-spacecraft wireless data interfacing. In *57th International Astronautical Congress*.

[Anand Karwa, 2015] Anand Karwa (2015). Google Brillo – An Internet Of Things OS That Runs on 32 MB RAM. http://trak.in/tags/business/2015/05/23/google-brillo-internet-of-things-operating-system/.

[Angius et al., 2012] Angius, F. et al. (2012). BLOOGO: BLOOm Filter Based GOssip Algorithm for Wireless NDN. In *Proc. of ACM NoM Workshop*, pages 25–30. ACM.

[Apache, 2016] Apache (2016). Apache Mynewt. https://mynewt.apache.org/.

[Arduino, 2016a] Arduino (2016a). Arduino. http://arduino.cc/.

[Arduino, 2016b] Arduino (2016b). Arduino Due. http://arduino.cc/en/Main/arduinoBoardDue.

[ARM mbed, 2015] ARM mbed (2015). mbed OS. https://mbed.org/technology/os/.

[Baar et al., 2008] Baar, M., Will, H., Blywis, B., Liers, A., Wittenburg, G., and Schiller, J. (2008). The ScatterWeb MSB-A2 Platform for Wireless Sensor Networks. Technical report, Freie Universität Berlin.

[Baccelli and Perkins, 2016] Baccelli, E. and Perkins, C. (2016). Multi-hop Ad Hoc Wireless Communication. https://tools.ietf.org/html/draft-ietf-intarea-adhoc-wireless-com-02. IETF Internet Draft.

[Baccelli and Schleiser, 2016] Baccelli, E. and Schleiser, K. (2016). Powering the Internet of Things with RIOT: Why? How? What is RIOT? *arXiv preprint arXiv:1603.03635*.

[Banks and Gupta, 2015] Banks, A. and Gupta, R. (2015). MQTT Version 3.1.1 Plus Errata 01. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html.

[Bao and Garcia-Luna-Aceves, 2001] Bao, L. and Garcia-Luna-Aceves, J. (2001). A New Approach to Channel Access Scheduling for Ad Hoc Networks. In *Proc. of the 7th annual international conference on Mobile computing and networking*, pages 210–221. ACM Press.

[Barry, 2012] Barry, R. (2012). FreeRTOS, a FREE open source RTOS for small embedded real time systems. http://www.freertos.org.

[Basiliere and Tully, 2014] Basiliere, P. and Tully, J. (2014). *Gartner Study: Makers and Startups Are the Ones Shaping the Internet of Things*. Maverick Research, Gartner.

[Beckmann and Dedi, 2015] Beckmann, K. and Dedi, O. (2015). sDDS: A portable data distribution service implementation for WSN and IoT platforms. In *Intelligent Solutions in Embedded Systems (WISES), 2015 12th International Workshop on*, pages 115–120. IEEE.

[Beekema, 2011] Beekema, M. (2011). *Fault-tolerant platform for intra-spacecraft modular wireless sensor network*. PhD thesis, TU Delft, Delft University of Technology.

[Bhatti et al., 2005] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. (2005). MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mobile Networks and Applications*, 10:563–579. Springer.

[Bierman et al., 2016] Bierman, A., Bjorklund, M., and Watsen, K. (2016). RESTCONF Protocol. `https://tools.ietf.org/html/draft-ietf-netconf-restconf-16`. IETF Internet Draft.

[Biswas et al., 2013] Biswas, T., Chakraborti, A., Ravindran, R., Zhang, X., and Wang, G. (2013). Contextualized information-centric home network. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 461–462. ACM.

[Blackberry Ltd., 2012] Blackberry Ltd. (2012). QNX. `http://www.qnx.com/`.

[BlackDuck, 2016] BlackDuck (2016). BlackDuck Open Hub code analysis of RIOT. `https://www.openhub.net/p/RIOT-OS`.

[Blackham, 2013] Blackham, B. (2013). *Towards Verified Microkernels for Real-Time Mixed-Criticality Systems*. PhD thesis, The University of New South Wales.

[Bluetooth SIG, 2016] Bluetooth SIG (2016). Bluetooth – Adopted Specifications. `https://www.bluetooth.com/specifications/adopted-specifications`.

[Blywis et al., 2010a] Blywis, B., Guenes, M., Juraschek, F., and Schiller, J. H. (2010a). Trends, advances, and challenges in testbed-based wireless mesh network research. *Mobile Networks and Applications*, 15(3):315–329. Springer.

[Blywis et al., 2010b] Blywis, B., Güneş, M., Juraschek, F., and Hofmann, S. (2010b). Gossip routing in wireless mesh networks. In *21st Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1572–1577. IEEE.

[Bormann, 2014] Bormann, C. (2014). 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400 (Proposed Standard). http://www.ietf.org/rfc/rfc7400.txt, IETF.

[Bormann et al., 2016] Bormann, C., Betzler, A., Gomez, C., and Demirkol, I. (2016). CoAP Simple Congestion Control/Advanced. IETF Internet Draft.

[Bormann et al., 2014] Bormann, C., Ersue, M., and Keranen, A. (2014). Terminology for constrained node networks. RFC 7228 (Informational). http://www.ietf.org/rfc/rfc7228.txt, IETF.

[Brandt et al., 2010] Brandt, A., Buron, J., and Porcu, G. (2010). Home Automation Routing Requirements in Low-Power and Lossy Networks. RFC 5826 (Informational). http://www.ietf.org/rfc/rfc5826.txt, IETF.

[Brorsson and Gunnarsson, 2016] Brorsson, J. and Gunnarsson, M. (2016). Compact Object Security for the Internet of Things. In *Master Thesis, Lund University*.

[Burke et al., 2013] Burke, J., Gasti, P., Nathan, N., and Tsudik, G. (2013). Securing instrumented environments over content-centric networking: the case of lighting control and NDN. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 394–398. IEEE.

[Campista et al., 2008] Campista, M. E. M., Esposito, P. M., Moraes, I. M., Passos, D. G., De Albuquerque, C. V. N., Saade, D. C. M., Rubinstein, M. G., et al. (2008). Routing metrics and protocols for wireless mesh networks. *IEEE network*, 22(1):6–12. IEEE.

[Cao et al., 2008] Cao, Q., Abdelzaher, T., Stankovic, J., and He, T. (2008). 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.

[Castellani et al., 2012] Castellani, A., Ministeri, G., Rotoloni, M., Vangelista, L., and Zorzi, M. (2012). Interoperable and globally interconnected Smart Grid using IPv6 and 6LoWPAN. In *Communications (ICC), 2012 IEEE International Conference on*, pages 6473–6478. IEEE.

[CCN-Lite, 2014] CCN-Lite (2014). CCN Lite: Lightweight implementation of the Content Centric Networking protocol. http://ccn-lite.net.

[Cerf and Cain, 1983] Cerf, V. G. and Cain, E. (1983). The DoD internet architecture model. *Computer Networks*, 7(5):307–318. Elsevier.

[Chapman, 2014] Chapman, A. (2014). Hacking into Internet Connected Light Bulbs. http://www.contextis.com/resources/blog/hacking-internet-connected-light-bulbs/.

[Chatzigiannakis et al., 2009] Chatzigiannakis, I., Fischer, S., Koninis, C., Mylonas, G., and Pfisterer, D. (2009). WISEBED: an open large-scale wireless sensor network testbed. In *International Conference on Sensor Applications, Experimentation and Logistics*, pages 68–87. Springer.

[ChibiOS, 2016] ChibiOS (2016). ChibiOS/RT. http://www.chibios.org/.

[Clausen and Jacquet, 2003] Clausen, T. and Jacquet, P. (2003). Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental). http://www.ietf.org/rfc/rfc3626.txt, IETF.

[Contiki, 2016] Contiki (2016). Contiki Operating System. http://www.contiki-os.org.

[CooCox, 2016] CooCox (2016). CooCox CoOS. http://www.coocox.org/.

[Cooprider et al., 2007] Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. (2007). Efficient Memory Safety for TinyOS. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 205–218. ACM Press.

[Cordero et al., 2013] Cordero, J., Yi, J., Clausen, T., and Baccelli, E. (2013). Enabling multihop communication in spontaneous wireless networks. *ACM SIGCOMM eBook on" Recent Advances in Networking*, 1:413–457. ACM Press.

[Corujo et al., 2012] Corujo, D., Aguiar, R. L., Vidal, I., and Garcia-Reinoso, J. (2012). A named data networking flexible framework for management communications. *IEEE Communications Magazine*, 50(12):36–43. IEEE.

[Daidone et al., 2014] Daidone, R., Dini, G., and Anastasi, G. (2014). On evaluating the performance impact of the IEEE 802.15. 4 security sub-layer. *Computer Communications*, 47:65–76. Elsevier.

[Dannewitz et al., 2013] Dannewitz, C. et al. (2013). Network of Information (NetInf): An information-centric networking architecture . *Computer Comm.*, 36(7):721 – 735. Elsevier.

[Decotognie and Pleinvevaux, 1993] Decotognie, J.-D. and Pleinvevaux, P. (1993). A Survey of Industrial Communication Networks. *Annals of Telecommunications*, 48(9-10):435–448. [invited paper].

[desvirt, 2016] desvirt (2016). DES-Virt: The DES Testbed virtualization framework. https://github.com/des-testbed/desvirt.

[Dezfouli et al., 2015] Dezfouli, B., Radi, M., Whitehouse, K., Razak, S. A., and Hwee-Pink, T. (2015). DICSA: Distributed and concurrent link scheduling algorithm for data gathering in wireless sensor networks. *Ad Hoc Networks*, 25:54–71. Elsevier.

[Dierks and Rescorla, 2008] Dierks, T. and Rescorla, E. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). http://www.ietf.org/rfc/rfc5246.txt, IETF.

[Dietrich et al., 2010] Dietrich, D., Bruckner, D., Zucker, G., and Palensky, P. (2010). Communication and Computation in Buildings: A Short Introduction and Overview. *Industrial Electronics, IEEE Transactions on*, 57(11):3577–3584. IEEE.

[Doherty et al., 2007] Doherty, L., Lindsay, W., and Simon, J. (2007). Channel-Specific Wireless Sensor Network Path Data. In *International Conference on Computer Communications and Networks (ICCCN)*. IEEE.

[Dohler et al., 2009] Dohler, M., Watteyne, T., Winter, T., and Barthel, D. (2009). Routing Requirements for Urban Low-Power and Lossy Networks. RFC 5548 (Informational). http://www.ietf.org/rfc/rfc5548.txt, IETF.

[Dong et al., 2010] Dong, W., Chen, C., Liu, X., and Bu, J. (2010). Providing OS support for wireless sensor networks: challenges and approaches. *Communications Surveys & Tutorials, IEEE*, 12(4):519–530. IEEE.

[Dunkels, 2001] Dunkels, A. (2001). Design and Implementation of the lwIP TCP/IP Stack. Technical report.

[Dunkels, 2003] Dunkels, A. (2003). Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98. ACM Press.

[Dunkels, 2011] Dunkels, A. (2011). The contikimac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science.

[Dunkels et al., 2011] Dunkels, A., Eriksson, J., Finne, N., and Tsiftes, N. (2011). Powertrace: Network-level power profiling for low-power wireless networks. Technical report, Swedish Institute of Computer Science.

[Dunkels et al., 2004] Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN*, pages 455–462. IEEE.

[Dunkels et al., 2006] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. (2006). Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Boulder, Colorado, USA.

[Duquennoy et al., 2015] Duquennoy, S., Nahas, B. A., Landsiedel, O., and Watteyne, T. (2015). Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys 2015)*, Seoul, South Korea. ACM.

[Durvy et al., 2008] Durvy, M., Abeillé, J., Wetterwald, P., O'Flynn, C., Leverett, B., Gnoske, E., Vidales, M., Mulligan, G., Tsiftes, N., Finne, N., et al. (2008). Making sensor networks IPv6 ready. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 421–422. ACM Press.

[Dutta et al., 2008] Dutta, P., Feldmeier, M., Paradiso, J., and Culler, D. (2008). Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 283–294. IEEE Computer Society.

[ebtables, 2016] ebtables (2016). ebtables: A filtering Tool for a Bridging Firewall. `http://ebtables.netfilter.org`.

[eCos, 2016] eCos (2016). eCos Embedded Operating System. `http://ecos.sourceware.org`.

[Ee et al., 2006] Ee, C. T., Fonseca, R., Kim, S., Moon, D., Tavakoli, A., Culler, D., Shenker, S., and Stoica, I. (2006). A modular network layer for sensorsets. In *Proc. of OSDI*, pages 249–262. USENIX Association.

[eLua, 2016] eLua (2016). eluaproject. `http://www.eluaproject.net/`.

[emul8, 2016] emul8 (2016). Emul8. `http://emul8.org/`.

[ERIKA, 2016] ERIKA (2016). ERIKA Enterprise. `http://erika.tuxfamily.org/drupal/`.

[Eriksson et al., 2007] Eriksson, J., Dunkels, A., Finne, N., Osterlind, F., and Voigt, T. (2007). Mspsim–an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, page 27.

[Ersue et al., 2015] Ersue, M., Romascanu, D., Schoenwaelder, J., and Herberg, U. (2015). Management of Networks with Constrained Devices: Problem Statement and Requirements. RFC 7547 (Informational). http://www.ietf.org/rfc/rfc7547.txt, IETF.

[Espruino, 2016] Espruino (2016). Espruino – JavaScript for Microcontrollers. http://www.espruino.com/.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131. ACM Press.

[Evanczuk, Stephen, 2013] Evanczuk, Stephen (2013). The most-popular MCUs ever. https://web.archive.org/web/20140703062337/http://edn.com/electronics-blogs/systems-interface/4419922/Slideshow--The-most-popular-MCUs-ever.

[Farrel et al., 2006] Farrel, A., Vasseur, J.-P., and Ash, J. (2006). A Path Computation Element (PCE)-Based Architecture. RFC 4655 (Informational). http://www.ietf.org/rfc/rfc4655.txt, IETF.

[Feeney and Nilsson, 2001] Feeney, L. M. and Nilsson, M. (2001). Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1548–1557. IEEE.

[Feeney et al., 2014] Feeney, L. M., Rohner, C., and Lindgren, A. (2014). How do the dynamics of battery discharge affect sensor lifetime? In *Wireless On-demand Network Systems and Services (WONS), 2014 11th Annual Conference on*, pages 49–56. IEEE.

[Feldmann, 2007] Feldmann, A. (2007). Internet clean-slate design: what and why? *ACM SIGCOMM Computer Communication Review*, 37(3):59–64. ACM Press.

[Ferrari et al., 2011] Ferrari, F., Zimmerling, M., Thiele, L., and Saukh, O. (2011). Efficient network flooding and time synchronization with glossy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 73–84. IEEE.

[Finn and Thubert, 2016] Finn, N. and Thubert, P. (2016). Deterministic Networking Problem Statement. https://tools.ietf.org/html/draft-ietf-detnet-problem-statement-08. IETF Internet Draft.

[Fonseca et al., 2008] Fonseca, R., Dutta, P., Levis, P., and Stoica, I. (2008). Quanto: tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 323–338. USENIX Association.

[Fotiou et al., 2012] Fotiou, N. et al. (2012). Illustrating a publish-subscribe Internet architecture. *Telecommunication Systems*, 51(4):233–245. Springer.

[FreakZ, 2013] FreakZ (2013). FreakZ – An open source Zigbee protocol stack for embedded platforms. https://sourceforge.net/projects/freakz/.

[Free Software Foundation, Inc., 2016] Free Software Foundation, Inc. (2016). lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip.

[FreeBSD, 2016] FreeBSD (2016). FreeBSD Ports. https://www.freebsd.org/-ports/.

[Freescale, 2015] Freescale (2015). The MC13224V SoC. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13224V.

[FTDI, 2010] FTDI (2010). FT232R USB UART IC Datasheet Version 2.09. http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf.

[Gavrilut et al., 2016] Gavrilut, D., Basaraba, R., and Cabau, G. (2016). Hackers Can Use Smart Sockets to Shut Down Critical Systems. https://labs.bitdefender.com/2016/08/hackers-can-use-smart-sockets-to-shut-down-critical-systems/.

[Gay et al., 2003] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not.*, 38(5):1–11. ACM Press.

[Ghodsi et al., 2012] Ghodsi, A. et al. (2012). Information-centric networking: Ready for the real world? *Dagstuhl Reports (Seminar 12361)*, 2(9):1–14.

[GNU, 2016] GNU (2016). GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[Gomez and Crowcroft, 2016] Gomez, C. and Crowcroft, J. (2016). TCP over Constrained-Node Networks. https://tools.ietf.org/html/draft-gomez-core-tcp-constrained-node-networks-00. IETF Internet Draft.

[Google, 2015] Google (2015). Project Brillo. https://developers.google.com/brillo/.

[Graham et al., 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM Press.

[Green Hills Software, 2015] Green Hills Software (2015). µ-velOSity. http://www.ghs.com/products/micro_velosity.html.

[GSM, 2016] GSM (2016). 3GPP specifications. http://www.3gpp.org/specifications.

[Gubbi et al., 2013] Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660. Elsevier.

[Gungor and Hancke, 2009] Gungor, V. C. and Hancke, G. P. (2009). Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *IEEE Transactions on Industrial Electronics*, 56(10):4258–4265. IEEE.

[Gura et al., 2004] Gura, N., Patel, A., Wander, A., Eberle, H., and Shantz, S. C. (2004). Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 119–132. Springer.

[Günes et al., 2014] Günes, M., Blywis, B., Frey, M., Hahm, O., Juraschek, F., Kumar, P., Mushtaq, Q., and Schleiser, K. (2008-2014). DES-Testbed. http://www.des-testbed.net. Homepage of the DES-Testbed.

[Hahm, 2007] Hahm, O. (2007). "Design and Implementation of a self-adjusting Time Synchronization in Wireless Sensor Networks". In *Diploma Thesis, Freie Universität Berlin*.

[Hail et al., 2015] Hail, M. A. M., Amadeo, M., Molinaro, A., and Fischer, S. (2015). On the Performance of Caching and Forwarding in Information-Centric Networking for the IoT. In *Wired/Wireless Internet Communications*, pages 313–326. Springer.

[HART Communication Foundation, 2008] HART Communication Foundation (2008). WirelessHART Specification 75: TDMA Data-Link Layer. HCF_SPEC-75.

[Härtig and Roitzsch, 2006] Härtig, H. and Roitzsch, M. (2006). Ten years of research on L4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*.

[Hartke, 2015] Hartke, K. (2015). Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641 (Proposed Standard). http://www.ietf.org/rfc/rfc7641.txt, IETF.

[Heer et al., 2011] Heer, T., Garcia-Morchon, O., Hummen, R., Keoh, S. L., Kumar, S. S., and Wehrle, K. (2011). Security Challenges in the IP-based Internet of Things. *Wireless Personal Communications*, 61(3):527–542. Springer.

[Henderson et al., 2008] Henderson, T. R., Lacage, M., Riley, G. F., Dowell, C., and Kopena, J. (2008). Network simulations with the ns-3 simulator. In *SIGCOMM demonstration*, volume 14.

[Hennessy and Patterson, 2003] Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers.

[Herberg et al., 2014] Herberg, U., Dearlove, C., and Clausen, T. (2014). Integrity Protection for the Neighborhood Discovery Protocol (NHDP) and Optimized Link State Routing Protocol Version 2 (OLSRv2). RFC 7183 (Proposed Standard). http://www.ietf.org/rfc/rfc7183.txt, IETF.

[Hochschule Offenburg, 2015] Hochschule Offenburg (2015). *Documentation of the emb6 Network Stack*, v0.1.0 edition. https://github.com/hso-esk/emb6/blob/b4ec037cd38c0f87013e3f0fb811f0f6da746f75/doc/pdf/emb6.pdf.

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

[Hoque et al., 2013] Hoque, M. et al. (2013). NLSR: Named-data Link State Routing Protocol. In *Proc. of ACM SIGCOMM WS on ICN*, pages 15–20. ACM.

[Huawei, 2015] Huawei (2015). Huawei LiteOS. http://www.huawei.com/minisite/iot/en/liteos.html.

[Huawei, 2015] Huawei (2015). Huawei Network Congress 2015 Announcement. http://pr.huawei.com/en/news/hw-432402-agilenetwork3.0.htm.

[Hughes et al., 2004] Hughes, B., Meier, R., Cunningham, R., and Cahill, V. (2004). Towards Real-time Middleware for Vehicular Ad Hoc Networks. In *Proceedings of the 1st ACM International Workshop on Vehicular Ad Hoc Networks*, VANET '04, pages 95–96. ACM Press.

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

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

[IEEE1588, 2014] IEEE1588 (2014). Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. http://www.nist.gov/el/isd/ieee/ieee1588.cfm.

[IEEE802.11, 2012] IEEE802.11 (2012). IEEE std. 802.11, Part. 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

[IEEE802.15.4, 2011] IEEE802.15.4 (2011). IEEE std. 802.15.4, Part. 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks.

[IEEE802.15.4e, 2012] IEEE802.15.4e (2012). IEEE802.15.4e-2012: IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer.

[IEEE802.15.6, 2008] IEEE802.15.6 (2008). 802.15.6 Call for Applications - Summary. https://mentor.ieee.org/802.15/documents?is_dcn=applications&is_group=0006.

[IIC, 2016] IIC (2016). Industrial Internet Consortium. http://www.iiconsortium.org/.

[Iima et al., 2009] Iima, Y., Kanzaki, A., Hara, T., and Nishio, S. (2009). Overhearing-based data transmission reduction for periodical data gathering in wireless sensor networks. In *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on*, pages 1048–1053. IEEE.

[INA121, 2000] INA121 (2000). INA121 FET-Input, Low Power, Instrumentation Amplifier. http://www.ti.com/litv/pdf/sbos078.

[IoT-LAB, 2016] IoT-LAB (2016). IoT-LAB: Very large scale open wireless sensor network testbed. https://www.iot-lab.info/hardware/m3/.

[IPSO, 2016] IPSO (2016). IPSO Alliance. http://www.ipso-alliance.org/.

[ITRON, 2016] ITRON (2016). ITRON project archive. http://www.ertl.jp/ITRON/home-e.html.

[Jacobson et al., 2009] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. (2009). Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM Press.

[Jedermann et al., 2014] Jedermann, R., Pötsch, T., and Lloyd, C. (2014). Communication techniques and challenges for wireless food quality monitoring. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2017). The Royal Society.

[JerryScript, 2016] JerryScript (2016). JavaScript engine for Internet of Things. http://jerryscript.net/.

[J.J. Garcia-Luna-Aceves and Rolando Menchaca-Mendez, 2012] J.J. Garcia-Luna-Aceves and Rolando Menchaca-Mendez (2012). STORM: A Framework for Integrated Routing, Scheduling, and Traffic Management in Ad Hoc Networks. *IEEE Transactions on Mobile Computing*, 11(8):1345–1357.

[Juraschek et al., 2013] Juraschek, F., Seif, S., and Güneş, M. (2013). Distributed Channel Assignment in Large-Scale Wireless Mesh Networks: A Performance Analysis. In *IEEE International Conference on Communications (ICC)*. IEEE.

[Kahn et al., 1999] Kahn, J. M., Katz, R. H., and Pister, K. S. (1999). Next century challenges: mobile networking for "Smart Dust". In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM Press.

[Karl and Willig, 2007] Karl, H. and Willig, A. (2007). *Protocols and architectures for wireless sensor networks*. John Wiley & Sons.

[Kellner et al., 2008] Kellner, S., Pink, M., Meier, D., and Blass, E.-O. (2008). Towards a Realistic Energy Model for Wireless Sensor Networks. In *The Fifth Annual Conference on Wireless On demand Network Systems and Services (WONS 2008)*, Garmisch-Partenkirchen, Germany. IEEE.

[Kent and Seo, 2005] Kent, S. and Seo, K. (2005). Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard). http://www.ietf.org/rfc/rfc4301.txt, IETF.

[Khan et al., 2016] Khan, S., Pathan, A.-S. K., and Alrajeh, N. A. (2016). *Wireless Sensor Networks: Current Status and Future Trends*. CRC Press.

[Kim et al., 2008] Kim, S. C., Kim, H., Song, J., Yu, M., and Mah, P. (2008). NanoQplus: A Multi-Threaded Operating System with Memory Protection Mechanism for WSNs. In *Proceedings of the CKWSN*, volume 20. IEEE.

[Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. (2009).

seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM Press.

[Ko et al., 2011] Ko, J., Eriksson, J., Tsiftes, N., Dawson-Haggerty, S., Vasseur, J., Durvy, M., Terzis, A., Dunkels, A., and Culler, D. (2011). Beyond Interoperability – Pushing the Performance of Sensor Network IP Stacks. In *Conference on Embedded Networked Sensor Systems (SenSys)*. ACM.

[Koponen et al., 2007] Koponen, T. et al. (2007). A Data-oriented (and Beyond) Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 37(4):181–192. ACM Press.

[Koster et al., 2016] Koster, M., Keranen, A., and Jimenez, J. (2016). Publish-Subscribe Broker for the Constrained Application Protocol (CoAP). `https://tools.ietf.org/html/draft-koster-core-coap-pubsub-05`. IETF Internet Draft.

[Kutscher et al., 2016] Kutscher, D., Eum, S., Pentikousis, K., Psaras, I., Corujo, D., Saucez, D., Schmidt, T., and Waehlisch, M. (2016). ICN Research Challenges. RFC 7927 (Informational). `http://www.ietf.org/rfc/rfc7927.txt`, IETF.

[L4, 2016] L4 (2016). Home of the L4 community. `http://l4hq.org/`.

[Landsiedel et al., 2004] Landsiedel, O., Wehrle, K., and Götz, S. (2004). Aeon: Accurate Prediction of Power Consumption in Sensor Networks. In *In Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-IIi)*. IEEE.

[Lenders, 2016] Lenders, M. (2016). Analysis and Comparison of Embedded Network Stacks. In *Master Thesis, Freie Universität Berlin*.

[Lenzen et al., 2015] Lenzen, C., Sommer, P., and Wattenhofer, R. (2015). PulseSync: An efficient and scalable clock synchronization protocol. *IEEE/ACM Transactions on Networking (TON)*, 23(3):717–727. IEEE.

[Levis, 2012] Levis, P. (2012). Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 207–220, Berkeley, CA, USA. USENIX Association.

[Levis et al., 2011] Levis, P., Clausen, T., Hui, J., Gnawali, O., and Ko, J. (2011). The Trickle Algorithm. RFC 6206 (Standards Track). `http://www.ietf.org/rfc/rfc6206.txt`, IETF.

[Levis et al., 2009] Levis, P. et al. (2009). Overview of existing routing protocols for low power and lossy networks. *IETF Internet Draft*.

[Levis et al., 2005] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. (2005). TinyOS: An Operating System for Sensor Networks. In Weber, W., Rabaey, J. M., and Aarts, E., editors, *Ambient Intelligence*, chapter 7, pages 115–148. Springer-Verlag.

[Li and Lazarou, 2004] Li, J. and Lazarou, G. Y. (2004). A Bit-map-assisted Energy-efficient MAC Scheme for Wireless Sensor Networks. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, IPSN '04, pages 55–60. ACM Press.

[Linear Technology, 2016] Linear Technology (2016). LTC4150 – Coulomb Counter/Battery Gas Gauge. http://www.linear.com/product/LTC4150.

[Liquorice, 2016] Liquorice (2016). Liquorice OS for Embedded Systems. http://sourceforge.net/p/liquorice/wiki/Home/.

[LK, 2016] LK (2016). LK embedded Kernel. https://github.com/littlekernel/lk.

[Lorien, 2016] Lorien (2016). Lorien. http://lorienos.sourceforge.net/index.php/Main_Page.

[Malkin and Minnear, 1997] Malkin, G. and Minnear, R. (1997). RIPng for IPv6. RFC 2080 (Proposed Standard). IETF.

[Mantis, 2016] Mantis (2016). MANTIS OS. http://www.sourceforge.net/projects/mantisos/.

[Martin Nieto, 2011] Martin Nieto, C. (2011). Virtualising a Wireless Network with KVM. In *Bachelor Thesis, Freie Universität Berlin*.

[Martocci et al., 2013] Martocci, J., Goyal, M., Philipp, M., Brandt, A., and Baccelli, E. (2013). Reactive Discovery of Point-to-Point Routes in Low-Power and Lossy Networks. RFC 6997 (Experimental). http://www.ietf.org/rfc/rfc6997.txt, IETF.

[Martocci et al., 2010] Martocci, J., Mil, P. D., Riou, N., and Vermeylen, W. (2010). Building Automation Routing Requirements in Low-Power and Lossy Networks. RFC 5867 (Informational). http://www.ietf.org/rfc/rfc5867.txt, IETF.

[Masmano et al., 2004] Masmano, M., Ripoll, I., Crespo, A., and Real, J. (2004). TLSF: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE.

[Mentor Graphics, 2015] Mentor Graphics (2015). Nucleus RTOS. http://www.mentor.com/embedded-software/nucleus/.

[Micrium, 2015a] Micrium (2015a). uC/OS-II. http://micrium.com/rtos/ucosii/overview/.

[Micrium, 2015b] Micrium (2015b). uC/OS-III. http://micrium.com/rtos/ucosiii/overview/.

[MicroPython, 2016] MicroPython (2016). Python for microcontrollers. https://micropython.org/.

[Microsoft, 2015] Microsoft (2015). Windows CE. http://microsoft.com/windowsce/.

[Milenković et al., 2006] Milenković, A., Otto, C., and Jovanov, E. (2006). Wireless sensor networks for personal health monitoring: Issues and an implementation. *Computer communications*, 29(13):2521–2533. Elsevier.

[Mills, 2006] Mills, D. (2006). Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. RFC 4330 (Informational). http://www.ietf.org/rfc/rfc4330.txt, IETF.

[Min et al., 2002] Min, R., Bhardwaj, M., Cho, S.-H., Ickes, N., Shih, E., Sinha, A., Wang, A., and Chandrakasan, A. (2002). Energy-centric enabling technologies for wireless sensor networks. *IEEE wireless communications*, 9(4):28–39. IEEE.

[Mirani, 2014] Mirani, L. (2014). Chip-makers are Betting that Moore's Law Won't Matter in the Internet of Things. http://qz.com/218514.

[Montenegro et al., 2007] Montenegro, G., Kushalnagar, N., Hui, J., and Culler, D. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard). {http://www.ietf.org/rfc/rfc4944.txt},IETF.

[Mosko and Tschudin, 2016] Mosko, M. and Tschudin, C. (2016). ICN "Begin-End" Hop by Hop Fragmentation. https://tools.ietf.org/html/draft-mosko-icnrg-beginendfragment-01. IRTF Internet Draft.

[Moubarak and Watfa, 2009] Moubarak, M. and Watfa, M. K. (2009). Embedded operating systems in wireless sensor networks. In *Guide to Wireless Sensor Networks*, pages 323–346. Springer.

[NanoRK, 2016] NanoRK (2016). NanoRK Operating System. http://www.nanork.org/.

[Nest, 2016a] Nest (2016a). OpenThread. https://github.com/openthread.

[Nest, 2016b] Nest (2016b). Thread Group. http://threadgroup.org.

[Netcraft, 2016] Netcraft (2016). February 2016 Web Server Survey. https://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html.

[nodeOS, 2016] nodeOS (2016). node OS. http://node-os.com/.

[NutOS, 2016] NutOS (2016). Nut/OS. http://www.ethernut.de/en/firmware/index.html.

[nuttx.org, 2015] nuttx.org (2015). NuttX Real-Time Operating System — NuttX Real-Time Operating System. http://nuttx.org. [Online; accessed 10-March-2015].

[OCF, 2016] OCF (2016). IoTivity. https://www.iotivity.org/.

[OMA LwM2M, 2016] OMA LwM2M (2016). OMA LightweightM2M v1.0. http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/oma-lightweightm2m-v1-0.

[OMG, 2014] OMG (2014). Data Distribution Service for Real-time Systems. http://www.omg.org/spec/DDS/.

[Open Handset Alliance, 2015] Open Handset Alliance (2015). Android Operating System. https://www.android.com/.

[OpenMote, 2016] OpenMote (2016). OpenMote-CC2538. http://www.openmote.com/hardware/openmote-cc2538-en.html.

[OpenWSN, 2016] OpenWSN (2016). Berkeley's OpenWSN Project. http://openwsn.berkeley.edu/.

[OpenZWave, 2016] OpenZWave (2016). OpenZWave. https://github.com/openzwave/.

[Orebaugh et al., 2006] Orebaugh, A., Ramirez, G., and Beale, J. (2006). *Wireshark & Ethereal network protocol analyzer toolkit*. Syngress.

[Ortmann, 2015] Ortmann, L. (2015). Virtualization of the RIOT Operating System. In *Diploma Thesis, Freie Universität Berlin*.

[Palattella et al., 2012] Palattella, M. R., Accettura, N., Dohler, M., Grieco, L. A., and Boggia, G. (2012). Traffic Aware Scheduling Algorithm for reliable low-power multi-hop IEEE 802.15. 4e networks. In *Personal Indoor and Mobile Radio Communications (PIMRC), 2012 IEEE 23rd Int. Symposium on*, pages 327–332. IEEE.

[Palattella et al., 2013a] Palattella, M. R., Accettura, N., Grieco, L. A., Boggia, G., Dohler, M., and Engel, T. (2013a). On optimal scheduling in duty-cycled industrial IoT applications using IEEE802.15.4e TSCH. *Sensors Journal, IEEE*, 13(10):3655–3666. IEEE.

[Palattella et al., 2013b] Palattella, M. R., Accettura, N., Vilajosana, X., Watteyne, T., Grieco, L. A., Boggia, G., and Dohler, M. (2013b). Standardized protocol stack for the internet of (important) things. *Communications Surveys & Tutorials, IEEE*, 15(3):1389–1406. IEEE.

[Pantelopoulos and Bourbakis, 2010] Pantelopoulos, A. and Bourbakis, N. G. (2010). A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(1):1–12. IEEE.

[Patel and Wang, 2010] Patel, M. and Wang, J. (2010). Applications, challenges, and prospective in emerging body area networking technologies. *IEEE Wireless Communications Magazine*, 17(1):80–88. IEEE.

[Payne, 2002] Payne, C. (2002). On the security of open source software. *Information systems journal*, 12(1):61–78. Wiley Online Library.

[Pebble, 2016] Pebble (2016). Pebble Smart Watch. `https://getpebble.com`.

[Perkins et al., 2003] Perkins, C., Belding-Royer, E., and Das, S. (2003). Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental). `http://www.ietf.org/rfc/rfc3561.txt`, IETF.

[Pister et al., 2009] Pister, K., Thubert, P., Dwars, S., and Phinney, T. (2009). Industrial Routing Requirements in Low-Power and Lossy Networks. RFC 5673 (Informational). `http://www.ietf.org/rfc/rfc5673.txt`, IETF.

[Pister and Doherty, 2008] Pister, K. S. J. and Doherty, L. (2008). TSMP: Time Synchronized Mesh Protocol. In *International Symposium on Distributed Sensor Networks (DSN)*, Orlando, Florida, USA. IASTED.

[powertop, 2011] powertop (2011). powertop -– Saving Power with Linux on Intel Platforms. `http://www.linuxpowertop.org/`.

[Quevedo et al., 2014] Quevedo, J., Corujo, D., and Aguiar, R. (2014). Consumer-driven information freshness approach for content centric networking. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 482–487. IEEE.

[Qumranet, 2006] Qumranet (2006). KVM: Kernel-based Virtualization Driver. http://www.linuxinsight.com/files/kvm_whitepaper.pdf.

[R, 2011] R (2011). The R Project for Statistical Computing. http://www.r-project.org/.

[Rajendran et al., 2006] Rajendran, V., Obraczka, K., and Garcia-Luna-Aceves, J. J. (2006). Energy-efficient, Collision-free Medium Access Control for Wireless Sensor Networks. *Wirel. Netw.*, 12(1):63–78. Springer-Verlag.

[Rakotoarivelo et al., 2010] Rakotoarivelo, T., Ott, M., Jourjon, G., and Seskar, I. (2010). OMF: a control and management framework for networking testbeds. *SIGOPS Oper. Syst. Rev.*, 43:54–59. ACM Press.

[Ramanathan, 1997] Ramanathan, S. (1997). A unified framework and algorithm for (T/F/C) DMA channel assignment in wireless networks. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 2, pages 900–907. IEEE.

[Rao et al., 2015] Rao, S., Chendanda, D., Deshpande, C., and Lakkundi, V. (2015). Implementing LWM2M in constrained IoT devices. In *Wireless Sensors (ICWiSe), 2015 IEEE Conference on*, pages 52–57. IEEE.

[Rawat et al., 2014] Rawat, P., Singh, K. D., Chaouchi, H., and Bonnin, J. M. (2014). Wireless sensor networks: a survey on recent developments and potential synergies. *The Journal of supercomputing*, 68(1):1–48. Springer.

[Raza et al., 2011] Raza, S., Duquennoy, S., Chung, T., Yazar, D., Voigt, T., and Roedig, U. (2011). Securing communication in 6LoWPAN with compressed IPsec. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–8. IEEE.

[Raza et al., 2013] Raza, S., Shafagh, H., Hewage, K., Hummen, R., and Voigt, T. (2013). Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors Journal*, 13(10):3711–3720. IEEE.

[Redwire Llc., 2015] Redwire Llc. (2015). Redwire Econotag II. http://redwire.myshopify.com/products/econotag-ii.

[Rescorla and Modadugu, 2012] Rescorla, E. and Modadugu, N. (2012). Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard). http://www.ietf.org/rfc/rfc6347.txt, IETF.

[Rhee et al., 2008] Rhee, I., Warrier, A., Aia, M., Min, J., and Sichitiu, M. L. (2008). Z-MAC: A Hybrid MAC for Wireless Sensor Networks. *IEEE/ACM Trans. Netw.*, 16(3):511–524. IEEE Press.

[Richard et al., 2005] Richard, C. et al. (2005). Defining an Optimal Active Route Timeout for the AODV Routing Protocol. In *Proc. of IEEE SECON*, pages 26–29. IEEE.

[RIOT, 2016a] RIOT (2016a). Arduino Sketch Support in RIOT. https://github.com/RIOT-OS/RIOT/tree/master/examples/arduino_hello-world.

[RIOT, 2016b] RIOT (2016b). List of RIOT contributors. https://github.com/RIOT-OS/RIOT/graphs/contributors.

[RIOT, 2016c] RIOT (2016c). RIOT Community License Discussion. https://github.com/RIOT-OS/RIOT/wiki/FAQ.

[RIOT, 2016d] RIOT (2016d). RIOT Community Processes. https://github.com/RIOT-OS/RIOT/wiki/RIOT-Community-Processes.

[RIOT, 2016e] RIOT (2016e). RIOT Task Forces. https://github.com/RIOT-OS/RIOT/wiki/Task-Forces.

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

[Rodrigo Muñoz, 2016] Rodrigo Muñoz, S. (2016). A scalable distributed autonomy system for fractionated satellite missions. Technical report.

[roll, 2016] roll (2016). Routing Over Low power and Lossy networks. https://datatracker.ietf.org/wg/roll.

[ROLL Mail Archive, 2016] ROLL Mail Archive (2016). non-storing mode vs storing mode. https://www.ietf.org/mail-archive/web/roll/current/msg09887.html.

[Rose, Karen and Eldridge, Scott and Chapin, Lyman, 2015] Rose, Karen and Eldridge, Scott and Chapin, Lyman (2015). *The Internet of Things: An Overview*. The Internet Society (ISOC).

[Rosenkranz et al., 2015] Rosenkranz, P., Wählisch, M., Baccelli, E., and Ortmann, L. (2015). A Distributed Test System Architecture for Open-source IoT Software. In *ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)*. ACM Press.

[RTEMS, 2016] RTEMS (2016). RTEMS – Real-Time Executive for Multiprocessor Systems. http://www.rtems.org.

[Saadallah et al., 2012] Saadallah, B. et al. (2012). CCNx for Contiki: implementation details. In *Tech. Report RT-0432*. INRIA.

[Saraswat and Yadav, 2010] Saraswat, L. and Yadav, P. S. (2010). A comparative analysis of wireless sensor network operating systems. *International Journal of Engineering and Technoscience*, 1(1):41–47.

[Schleiser, 2016a] Schleiser, K. (2016a). Murdock—A simple CI server written in Python. https://github.com/kaspar030/murdock.

[Schleiser, 2016b] Schleiser, K. (2016b). Murdock—CI Frontend. https://ci.riot-labs.de/.

[Schmidt et al., 2007] Schmidt, D., Krämer, M., Kuhn, T., and Wehn, N. (2007). Energy modelling in sensor networks. *Advances in Radio Science*, 3(5):347–351. Copernicus Publications.

[Schmidt et al., 2016] Schmidt, T. C., Wölke, S., Berg, N., and Wählisch, M. (2016). Let's collect names: How PANINI limits FIB tables in name based routing. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 458–466. IFIP.

[Schoenwaelder, 2012] Schoenwaelder, J. (2012). Translation of Structure of Management Information Version 2 (SMIv2) MIB Modules to YANG Modules. RFC 6643 (Proposed Standard). http://www.ietf.org/rfc/rfc6643.txt, IETF.

[SCIOPTA Systems AG, 2015] SCIOPTA Systems AG (2015). SCIOPTA. http://www.sciopta.com/products/kernel.html.

[Security Space, 2015] Security Space (2015). Mail Server Survey. http://www.securityspace.com/s_survey/data/man.201504/mxsurvey.html.

[Segger, 2015] Segger (2015). embOS. https://www.segger.com/embos.html.

[Selander et al., 2016] Selander, G., Mattsson, J., Palombini, F., and Seitz, L. (2016). Object Security of CoAP (OSCOAP). IETF Internet Draft.

[Sethi et al., 2016] Sethi, M., arkko, J., Keranen, A., and Back, H.-M. (2016). Practical Considerations and Implementation Experiences in Securing Smart Object Networks. https://tools.ietf.org/html/draft-aks-lwig-crypto-sensors-01. IETF Internet Draft.

[Seward et al., 2004] Seward, J., Nethercote, N., and Fitzhardinge, J. (2004). Valgrind, an open-source memory debugger for x86-GNU/Linux. In *UKUUG Linux Developers' Conference*.

[Shafer, 2011] Shafer, P. (2011). An Architecture for Network Management Using NETCONF and YANG. RFC 6244 (Informational). http://www.ietf.org/rfc/rfc6244.txt, IETF.

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

[Shelby et al., 2014] Shelby, Z., Hartke, K., and Bormann, C. (2014). The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard). http://www.ietf.org/rfc/rfc7252.txt, IETF.

[Shelby et al., 2016] Shelby, Z., Koster, M., Bormann, C., and van der Stok, P. (2016). CoRE Resource Directory. http://tools.ietf.org/html/draft-ietf-core-resource-directory-08. IETF Internet Draft.

[Shnayder et al., 2004] Shnayder, V., Hempstead, M., Chen, B., Werner-Allen, G., and Wels, M. (2004). Simulating the power consumption of large-scale sensor network applications. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200. ACM Press.

[Shrestha et al., 2007] Shrestha, S., Shrestha, S., Lee, A., Lee, J., Seo, D.-W., Lee, K., Lee, J., Chong, S., and Myung, N. H. (2007). A Group of People Acts like a Black Body in a Wireless Mesh Network. In Lee, A., editor, *Proc. IEEE Global Telecommunications Conference GLOBECOM '07*, pages 4834–4839. IEEE.

[Simunic et al., 1999] Simunic, T., Benini, L., Benini, S. L., and Micheli, G. D. (1999). Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. In *in Proc. Design Automation Conf*, pages 867–872. ACM Press.

[SOS, 2016] SOS (2016). SOS 2.X. https://projects.nesl.ucla.edu/public/sos-2x.

[ST Microelectronics, 2015] ST Microelectronics (2015). STM32F100VC Microcontroller. http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN775/PF216851.

[Stanford-Clark and Truong, 2014] Stanford-Clark, A. and Truong, H. L. (2014). MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2. http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf.

[Stanislowski et al., 2014] Stanislowski, D., Vilajosana, X., Wang, Q., Watteyne, T., and Pister, K. (2014). Adaptive Synchronization in IEEE802.15.4e Networks. *IEEE Transactions on Industrial Informatics*, 10(1):795–802. IEEE.

[Stathopoulos et al., 2008] Stathopoulos, T., McIntire, D., and Kaiser, W. J. (2008). The Energy Endoscope: Real-Time Detailed Energy Accounting for Wireless Sensor Nodes. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 383–394, Washington, DC, USA. IEEE Computer Society.

[Stouffer et al., 2011] Stouffer, K. A., Falco, J. A., and Scarfone, K. A. (2011). SP 800-82. Guide to Industrial Control Systems (ICS) Security: Supervisory Control and Data Acquisition (SCADA) Systems, Distributed Control Systems (DCS), and Other Control System Configurations Such As Programmable Logic Controllers (PLC). Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States.

[Strazdins et al., 2010] Strazdins, G., Elsts, A., and Selavo, L. (2010). Mansos: easy to use, portable and resource efficient operating system for networked embedded devices. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 427–428. ACM Press.

[Sun et al., 2010] Sun, R., Guo, J., and Gill, E. (2010). Opportunities and Challenges of Wireless Sensor Networks in Space. In *61st International Astronautical Congress*, pages 1–7. International Astronautical Federation.

[SYSGO, 2016] SYSGO (2016). PikeOS. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[Tan et al., 2002] Tan, T. K., Raghunathan, A., and Jha, N. K. (2002). EMSIM: An Energy Simulation Framework for an Embedded Operating System. In *IEEE International Symposium on Circuits and Systems*, pages 464–467. IEEE Computer Society.

[TARWIS, 2011] TARWIS (2011). TARWIS. http://www.wisebed.eu/index.php/testbeddescripton/130-gettingstarted-tarwis.

[tc, 2016] tc (2016). tc (traffic control). http://linux.die.net/man/8/tc.

[Tecmint, 2014] Tecmint (2014). 30 Big Companies and Devices Running on GNU/Linux. http://www.tecmint.com/big-companies-and-devices-running-on-gnulinux/.

[tessel, 2016] tessel (2016). Tessel Embedded Development Platform. https://tessel.io/.

[ThreadX, 2016] ThreadX (2016). ThreadX. http://rtos.com/products/threadx/.

[TI CC3000, 2015] TI CC3000 (2015). Texas Instruments cc3000. http://www.ti.com/product/cc3000.

[Tinka et al., 2010] Tinka, A., Watteyne, T., and Pister, K. (2010). A decentralized scheduling algorithm for time synchronized channel hopping. In *Ad Hoc Networks*, pages 201–216. Springer.

[Tschofenig et al., 2015] Tschofenig, H., Arkko, J., Thaler, D., and McPherson, D. (2015). Architectural Considerations in Smart Object Networking. RFC 7452 (Informational). http://www.ietf.org/rfc/rfc7452.txt, IETF.

[ucLinux, 2016] ucLinux (2016). Embedded Linux/Microcontroller Project. http://www.uclinux.org.

[van der Heijden and van der Mullen, 2002] van der Heijden, H. and van der Mullen, J. (2002). General treatment of the interplay between fluid and radiative transport phenomena in symmetric plasmas: the sulphur lamp as a case study. *Journal of Physics D: Applied Physics*, 35(17):2112. IOP Publishing.

[van der Stok and Bierman, 2016] van der Stok, P. and Bierman, A. (2016). CoAP Management Interface. https://tools.ietf.org/html/draft-vanderstok-core-comi-09. IETF Internet Draft.

[Vasseur, 2014] Vasseur, J. (2014). Terms Used in Routing for Low-Power and Lossy Networks. RFC 7102 (Informational). http://www.ietf.org/rfc/rfc7102.txt, IETF.

[Vasseur et al., 2012] Vasseur, J., Kim, M., Pister, K., Dejean, N., and Barthel, D. (2012). Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks. RFC 6551 (Proposed Standard). http://www.ietf.org/rfc/rfc6551.txt, IETF.

[Veillette et al., 2016] Veillette, M., Pelov, A., Somaraju, A., Turner, R., and Minaburo, A. (2016). CBOR Encoding of Data Modeled with YANG. https://tools.ietf.org/html/draft-ietf-core-yang-cbor-02. IETF Internet Draft.

[W3Cook, 2016] W3Cook (2016). OS Market Share and Usage Trends. http://www.w3cook.com/os/summary/.

[Wählisch et al., 2013] Wählisch, M., Schmidt, T. C., and Vahlenkamp, M. (2013). Backscatter from the data plane–threats to stability and security in information-centric network infrastructure. *Computer Networks*, 57(16):3192–3206. Elsevier.

[Waldrop, 2016] Waldrop, M. M. (2016). The chips are down for Moore's law. *Nature*, 530(7589):144–147. Nature Publishing Group.

[Walls, 2013] Walls, C. (2013). Dynamic memory and heap continuity. http://www.embedded.com/design/programming-languages-and-tools/4416457/EMB-tm-6-15-13-Dynamic-memory-and-heap-contiguity.

[Wang et al., 2012] Wang, L. et al. (2012). OSPFN: An OSPF Based Routing Protocol for Named Data Networking. Technical report, University of Memphis and University of Arizona.

[Wang et al., 2006] Wang, Q., Hempstead, M., and Yang, W. (2006). A Realistic Power Consumption Model for Wireless Sensor Network Devices. In *SECON '06: 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks*, pages 286–295. IEEE Computer Society.

[Watteyne et al., 2009] Watteyne, T., Mehta, A., and Pister, K. (2009). Reliability Through Frequency Diversity: Why Channel Hopping Makes Sense. In *Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks (PE-WASUN)*, pages 116–123. ACM.

[Watteyne et al., 2015] Watteyne, T., Palattella, M., and Grieco, L. (2015). Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement. RFC 7554 (Informational). http://www.ietf.org/rfc/rfc7554.txt, IETF.

[Wehrle et al., 2010] Wehrle, K., Günes, M., and Gross, J., editors (2010). *Modeling and Tools for Network Simulation*. Springer, 1st edition.

[West and Dedrick, 2001] West, J. and Dedrick, J. (2001). Open source standardization: the rise of Linux in the network era. *Knowledge, Technology & Policy*, 14(2):88–112. Springer.

[Will et al., 2009] Will, H., Schleiser, K., and Schiller, J. H. (2009). A real-time kernel for wireless sensor networks employed in rescue scenarios. In *IEEE LCN*. IEEE.

[Wind River, 2016] Wind River (2016). Wind River Rocket. http://www.windriver.com/products/operating-systems/rocket/.

[Wind River Systems, 2015] Wind River Systems (2015). VxWorks. http://www.windriver.com/products/vxworks/.

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

[Xie et al., 2010] Xie, W. et al. (2010). A Performance Analysis of Point-to-Point Routing along a Directed Acyclic Graph in Low Power and Lossy Networks. In *Proc. of IEEE NBiS*, pages 111–116. IEEE.

[Z-Wave, 2016] Z-Wave (2016). Z-Wave Smart Home. http://www.z-wave.com/.

[Zephyr, 2016] Zephyr (2016). Zephyr Project. https://www.zephyrproject.org/.

[Zhu and Corson, 2001] Zhu, C. and Corson, M. S. (2001). A five-phase reservation protocol (FPRP) for mobile ad hoc networks. *Wireless networks*, 7(4):371–384. Springer.

[ZigBee Alliance, 2016] ZigBee Alliance (2016). ZigBee. http://www.zigbee.org/.

[Zolertia, 2015] Zolertia (2015). Z1 Datasheet. http://www.zolertia.com/.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Internet des Objets: Bases Logicielles et Protocoles Réseaux pour un Déploiement Universel Efficace en Energie

Mots clefs : Internet des Objets, systèmes embarqués, réseaux san fil

Résumé : L'Internet des Objets a pour but d'intégrer des milliards d'objets connectés dans l'Internet. Du point de vue matériel, ces objets connectés sont de petits ordinateurs très bon marché, basés sur des microcontrôleurs et des puces radio efficaces en énergie apparus récemment, couplés avec des capteurs et actionneur divers, le tout alimenté par une batterie de très petite taille. Ces objets connectés sont donc typiquement très contraints en ressources telles que CPU, mémoire et énergie. De plus, les liens radio à travers lesquels communiquent les objets connectés ont une capacité très limitée, sont souvent sujet à des taux de perte importants, et peuvent requérir du routage spontané entre objets connectés pour fournir la connectivité nécessaire. Ces caractéristiques posent des défis, d'une part en termes de logiciel embarqué s'exécutant sur les objets connectés, et d'autre part en termes de protocols réseaux utilisés par les objets connectés pour communiquer. En conséquence, de nouvelles méthodes et outils expérimentaux sont nécessaires pour étudier in vivo les réseaux formés d'objets connectés, de nouvelles plateformes logicielles sont nécessaires pour exploiter efficacement les objets connectés, et des protocoles de communication innovants sont nécessaire pour interconnecter ces objets. La présente thèse relève en partie ces défis, en introduisant des nouveaux outils facilitant l'utilisation de grands réseaux test interconnectant de nombreux objets connectés, un nouveau système d'exploitation (RIOT) utilisable sur une très grande variété d'objets connectés, ainsi que plusieurs nouveaux mécanismes utilisant le paradigme des réseaux centrés contenus pour améliorer significativement l'efficacité énergétique des protocoles de communication standards de l'Internet des Objets.

Title: Enabling Energy Efficient Smart Object Networking at Internet-Scale

Keywords: Internet of Things, embedded systems, wireless networks

Abstract: The Internet of Things aims to seamlessly integrate billions of so-called Smart Objects into traditional Internet infrastructures. From the hardware perspective, Smart Objects emerged when tiny, cheap computers became available, combining energy efficient micro-controllers, low-power radio transceivers, and sensors as well as actuators interacting with the physical world, often powered by batteries. Typically, Smart Objects are thus heavily constrained in terms of CPU, memory and energy resources. Furthermore, wireless links used for communication among Smart Objects or towards the Internet are often slow, subject to high packet loss, and may require spontaneous store-and-forward among peer Smart Objects to ensure connectivity.
Such characteristics pose challenges, on one hand in terms of software running on Smart Objects, and on the other hand in terms of network protocols Smart Objects use to communicate.
In consequence, novel evaluation methods and experimental tools are needed to study Smart Object networks in vivo, new software platforms are needed to efficiently operate Smart Objects, and innovative networking paradigms and protocols are required to interconnect Smart Objects.
This thesis addresses these challenges by introducing new tools for large scale testbed-driven experimental research, a novel operating system (RIOT) applicable to a wide variety of connected Smart Objects, and several new mechanisms leveraging information-centric networking which significantly improve energy-efficiency compared to state-of-the-art network protocols in the Internet of Things.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France