Diss. ETH No. 22398

Scalable Web Technology for the Internet of Things

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Frank Matthias Kovatsch
Dipl.-Ing., Friedrich-Alexander-Universität Erlangen-Nürnberg
born on 2 September 1982
citizen of Germany

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Carsten Bormann, co-examiner
Prof. Dr. Thiemo Voigt, co-examiner

2015

Abstract

The Internet of Things (IoT) can be considered as a modern manifestation of Mark Weiser's classic vision of ubiquitous computing where tiny networked computers become part of everyday objects interweaving the virtual world and the physical world. The concept of the IoT originated some 15 years ago from linking real-world artifacts to virtual counterparts through radio-frequency identification (RFID) tags. More recently, environments have become 'smart' by augmenting physical objects with sensing or actuation capabilities and networking them with digital services. The ongoing standardization of Internet protocols for such IoT devices enables the seamless integration of smart things into the Internet. This trend is expected to eventually result in hundreds of billions of connected devices that need to be programmed, managed, and maintained. It has been shown that Web technology can significantly ease this process by providing well-known patterns and tools for developers and users. The existing solutions are, however, often too heavyweight for highly resource-constrained IoT devices. Indeed, most connected devices are expected to remain resource-constrained, as progress in technology witnessed by Moore's Law is primarily leveraged to minimize dimensions, power consumption, and unit costs.

This dissertation presents a comprehensive solution for the seamless integration of highly resource-constrained IoT systems into the World Wide Web. Our thesis is that *existing protocols and programming models do not effectually meet the needs of the IoT. We identify two key challenges for the vision to succeed: application-layer interoperability and improved usability for both developers and users. Both requirements can be met by an approach that amalgamates results from the field of Wireless Sensor Networks and the World Wide Web.* This leads to the research questions (i) how to scale Web technology down to resource-constrained devices, (ii) how to scale it up to hundreds of billions of devices, and (iii) how to use it to improve the usability of the tiny networked computers. Our work addresses the resulting challenges with the following contributions: Being actively involved in the design and standardization of the Constrained Application Protocol (CoAP) within the Internet Engineering Task Force (IETF), we (i) evaluate the new Web protocol

in the different components of IoT systems, namely resource-constrained devices, Cloud-based services, and user interaction. Based on this, we (ii) propose system architectures and guidelines for an optimal implementation and utilization of CoAP. Furthermore, we (iii) present concepts and tools for Web-like software development for the IoT. To support our thesis, we also (iv) provide working open source implementations of our concepts, which build the basis for several IoT projects in academia and industry.

More concretely, we show in this dissertation that the CoAP protocol suite closes the technological gap between low-power IoT devices and the well-known patterns of the Web. We first consider resource-constrained environments, where efficient Web technology can relieve application developers from the burdens of embedded programming while maintaining the performance of classic approaches. In addition to a proof of concept and system evaluation, we give guidelines that allow for significantly smaller memory footprints of CoAP implementations. Next, we show that the low overhead of the new protocol also improves performance in unconstrained environments, such as IoT cloud services that have to manage the myriad of IoT devices. We present a system architecture for scalable back-end services that outperforms classic high-performance Hypertext Transfer Protocol (HTTP) Web servers as well as other state-of-the-art CoAP implementations. Finally, our work evaluates usability aspects of the Web programming model for IoT applications. We show that Web mashups, that is, the linking of different services through lightweight scripting, are also directly applicable to our concepts for highly resource-constrained systems. Complementary, we study Web browser support for CoAP to fully close the gap between IoT devices and the Web. Based on these findings, we motivate the design decisions behind CoAP, in particular for our contributions, and explain how developers can improve their protocol implementations accordingly.

Along with this dissertation, we deliver open source implementations of our approach that go beyond prototypes. Our *Erbium (Er) REST Engine* is an optimized CoAP implementation for constrained environments. It provides application developers with resource handler abstractions like regular Web frameworks while maintaining a small memory footprint. Erbium became the default CoAP implementation for Contiki, a popular embedded operating system for the IoT that is used in many industry products. Our Java-based *Californium (Cf) CoAP framework* enables high-performance CoAP services. The project also contains a DTLS 1.2 implementation called *Scandium (Sc)* and the *Actinium (Ac)* application server, a RESTful runtime system for Web-like IoT mashups. Representing the state of the art for RESTful IoT services, the Californium project was adopted by the Eclipse Foundation within its IoT Industry Working Group.

Kurzfassung

Der Begriff *Internet der Dinge*, im Englischen Internet of Things (IoT), beschreibt die Vision, die virtuelle mit der realen Welt eng zu verquicken. Hierbei handelt es sich um eine Konkretisierung des Ubiquitous Computings, welches von Mark Weiser zu Beginn der 1990er Jahre geprägt wurde. Die Idee zum IoT entstand vor etwa 15 Jahren, als die RFID-Technologie aufkam und dazu verwendet wurde, Alltagsgegenstände per Marker mit virtuellen Abbildern in IT-Systemen zu koppeln. Inzwischen ist es möglich, Dinge direkt mit computergestützter Intelligenz auszustatten, die zwar nur über begrenzte Ressourcen verfügt, aber dennoch in der Lage ist, drahtlos zu kommunizieren, physische Zustände zu messen und Aktoren zu steuern. Die aktuelle Standardisierung neuer Internetprotokolle erlaubt es überdies, derartige IoT-Geräte nahtlos mit dem Internet zu verbinden. Es wird erwartet, dass diese Entwicklung zur Vernetzung von Hunderten Milliarden Geräten führen wird, die allesamt programmiert, verwaltet und gewartet werden müssen. Aktuelle Untersuchungen zeigen, dass die Web-Technologie entscheidend zu diesem Prozess beitragen kann, da deren bewährte Muster und Werkzeuge die Arbeit für Entwickler und Benutzer vereinfachen. Die bisherigen Lösungen sind für die knappen Ressourcen gängiger IoT-Geräte jedoch meist zu schwergewichtig, zumal sich die Leistung vieler Geräte nicht wesentlich ändern dürfte. Der Grund hierfür ist, dass der stete Fortschritt im Sinne des mooreschen Gesetzes im Rahmen des IoT auch weiterhin hauptsächlich zur Minimierung des Formfaktors, des Energieverbrauchs und vor allem der Kosten genutzt werden dürfte.

Die vorliegende Dissertation liefert eine umfassende Lösung für das Problem, IoT-Geräte mit begrenzten Ressourcen nahtlos in das *World Wide Web* zu integrieren. Unsere These lautet, dass *vorhandene Protokolle und Programmiermodelle den Anforderungen des IoT nicht genügen, da für eine praktikable Umsetzung Interoperabilität und Software-Ergonomie im Vordergrund stehen müssen. Diese Anforderungen können jedoch erfüllt werden, indem man die Resultate aus den Forschungsbereichen der drahtlosen Sensornetze und des World Wide Webs geeignet kombiniert.* Daraus ergeben sich sich die folgenden Forschungsfragen: (i) Wie kann Web-Technologie auf ressourcenarme Geräte herunterskaliert werden? (ii) Wie kann sie auf Hunderte Milliarden Geräte hochskaliert werden? (iii) Wie kann man sie nutzen, um die Software-Ergonomie von vernetzten eingebetteten Systemen zu verbessern? Einen wichtigen Schritt stellt hierbei das neue

Constrained Application Protocol (CoAP) dar, an dessen Entwicklung wir im Rahmen dieser Arbeit aktiv beteiligt sind. CoAP wurde explizit für ressourcenarme Geräte und zugleich Maschine-zu-Maschine-Kommunikation innerhalb der Internet Engineering Task Force (IETF) standardisiert. Dabei leisten wir mit dieser Arbeit die folgenden Beiträge: Wir (i) evaluieren das neue Protokoll in den entsprechenden Systembereichen des IoT, nämlich ressourcenarme Geräte, cloud-basierte Dienste und Benutzerinteraktion. Dies ermöglicht uns den (ii) Entwurf von fundierten Systemarchitekturen sowie Richtlinien für CoAP. Des Weiteren entwickeln wir (iii) Konzepte und Hilfsmittel für einen Web-ähnlichen Softwareentwicklungsprozess im IoT. Wir untermauern unsere These mit (iv) funktionsfähigen Open-Source-Implementierungen unserer Konzepte, welche bereits von einigen IoT-Projekten in Wissenschaft und Industrie verwendet werden.

In der vorliegenden Dissertation weisen wir im Detail nach, wie CoAP und seine Erweiterungen die technische Lücke zwischen IoT-Geräten mit knappen Ressourcen und bewährter Web-Technologie schliesst. Zunächst zeigen wir, wie effizient umgesetzte Muster aus der Web-Welt die Entwicklung von vernetzten eingebetteten Systemen ohne Leistungseinbussen vereinfachen kann. Wir evaluieren unser Konzept anhand eines Prototypen, dessen weitere Optimierung zu einer stabilen Implementierung und einem Leitfaden zur Speicheroptimierung geführt hat. Im Anschluss zeigen wir, dass der geringe Overhead von CoAP auch im Bereich der Cloud-basierten IoT-Dienste von Vorteil ist, da er es ermöglicht, die grosse Zahl an erwarteten Geräten zu bewältigen. Hierzu präsentieren wir eine performante und skalierbare Systemarchitektur für das IoT-Service-Backend, welche sowohl aktuelle Hochleistungs-HTTP-Server als auch andere CoAP-Lösungen leistungsmässig übertrifft. Schliesslich betrachten wir die Benutzerfreundlichkeit für IoT-Entwickler, die sich aus unserer Web-basierten Lösung ergibt. Wir zeigen, dass Web-Mashups zur Verknüpfung mehrerer Dienste durch einfaches Scripting direkt auf unsere Konzepte für ressourcenarme IoT-Geräte anwendbar sind. Ergänzend analysieren wir die Vorteile durch die Unterstützung von CoAP im Web-Browser. Anhand dieser Erkenntnisse motivieren wir auch die Entwurfsentscheidungen, die bezüglich CoAP getroffen wurden, und verdeutlichen, wie Entwickler die Protokollspezifikation am besten umsetzen können.

Im Rahmen dieser Dissertation stellen wir auch umfangreiche Open-Source-Implementierungen zur Verfügung: Unsere *Erbium (Er) REST Engine* ist eine CoAP-Implementierung für eingebettete Systeme mit geringen Ressourcen, welche Aspekte der Software-Ergonomie berücksichtigt. Erbium ist Teil von Contiki, einem weit verbreitetem Betriebssystem für IoT-Geräte, das auch in Industrieprodukten verwendet wird. Das Java-basierte *Californium (Cf) CoAP Framework* ist für skalierbare Dienste in der Cloud gedacht. Dieses Eclipse-Projekt umfasst eine umfangreiche und leistungsstarke CoAP-Implementierung, Sicherheit durch die *Scandium (Sc)* DTLS-1.2-Implementierung und unsere *Actinium (Ac)* Laufzeitumgebung für IoT-Mashups.

Contents

Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks.

6TiSCH IPv6 over the TSCH mode of IEEE 802.15.4e.

Ac Actinium.

ACE Authentication and Authorization for Constrained Environments.

ACK Acknowledgement.

ADC analog-to-digital converter.

AJAX Asynchronous JavaScript and XML.

AMPED Asynchronous Multi-Process Event-Driven.

API application programming interface.

BLE Bluetooth Low Energy.

CBOR Concise Binary Object Representation.

Cf Californium.

CLI command line interface.

CoAP Constrained Application Protocol.

CoCoA CoAP Congestion Control Advanced.

CON Confirmable.

CoRE Constrained RESTful Environments.

Cu Copper.

DECT ULE DECT ultra low energy.

DPWS Devices Profile for Web Services.

DTLS Datagram Transport Layer Security.

EBHTTP Embedded Binary HTTP.

ECC elliptic curve cryptography.

EPC Electronic Product Code.

Er Erbium.

ETSI European Telecommunication Standards Institute.

EUI-64 64-bit Extended Unique Identifier.

EXI Efficient XML Interchange.

GPIO general-purpose input/output.

GUI graphical user interface.

HATEOAS Hypermedia as the Engine of Application State.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IANA Internet Assigned Numbers Authority.

IDE integrated development environment.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

IPSO IP for Smart Objects.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

JVM Java Virtual Machine.

LLN low-power lossy network.

LWM2M OMA Lightweight M2M.

M2M machine-to-machine.

MEMS microelectromechanical systems.

MID message identifier.

MP Multi-Process.

MPL Multicast Protocol for Low power and Lossy Networks.

MQTT Message Queuing Telemetry Transport.

MT Multi-Threaded.

MTU maximum transmission unit.

NAT network address translation.

NON Non-confirmable.

OMA Open Mobile Alliance.

OS operating system.

OWL Web Ontology Language.

PIPELINED Multi-Threaded Pipelined.

RD resource directory.

RDC radio duty cycling.

REST Representational State Transfer.

RFID radio-frequency identification.

ROA resource-oriented architecture.

RPL IPv6 Routing Protocol for Low-Power and Lossy Networks.

RST Reset.

RTO retransmission timeout.

RTT round-trip delay time.

SaaS Software as a Service.

Sc Scandium.

SCADA Supervisory Control and Data Acquisition.

SEDA Staged Event-Driven Architecure.

SMS Short Message Service.

SoC system on a chip.

SPED Single-Process Event-Driven.

SYMPED Symmetric Multi-Processor Event-Driven.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TSCH time slotted channel hopping.

UART universal asynchronous receiver/transmitter.

UDP User Datagram Protocol.

URI Uniform Resource Identifier.

WoT Web of Things.

WSN wireless sensor network.

WSNs Wireless Sensor Networks.

XHR XMLHttpRequest.

XUL XML User Interface Language.

Chapter 1

Introduction

In its general sense, the Internet of Things (IoT) describes the concept of interconnecting the virtual world of computers with the real world of physical artifacts [130]. The term was coined in 1999 by the Auto-ID Center at the Massachusetts Institute of Technology (MIT), which has been active in the field of networked RFID and emerging sensing technologies [30, 165]. Sanjay Sarma, David L. Brock, and Kevin Ashton envisioned physical objects acting as nodes in a networked physical world by electronically tagging the objects. This was realized through the Electronic Product Code (EPC), which can be attached to everyday objects through RFID tags. It enables efficient referencing to pertinent information or digital representations in computer systems, thereby linking the physical object to the virtual world.

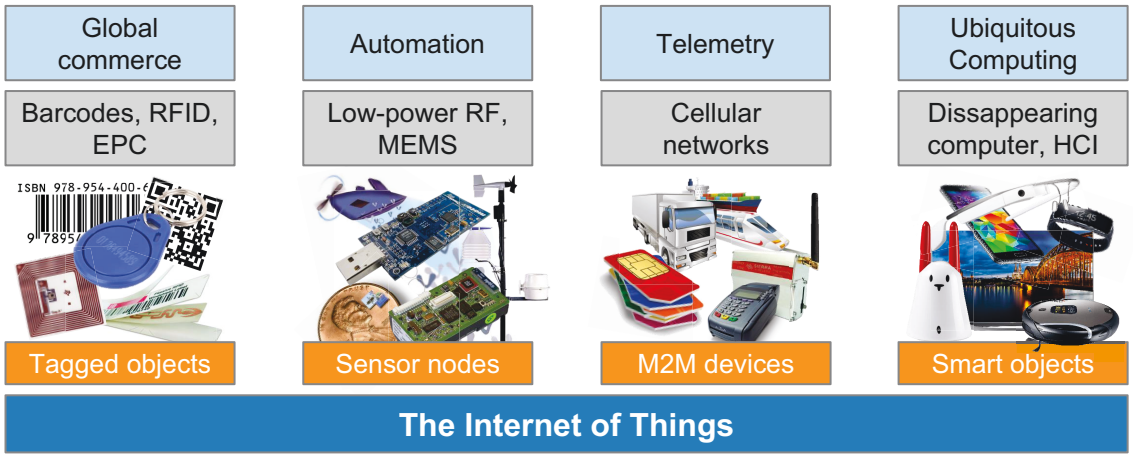


Figure 1.1: The concept today called the Internet of Things (IoT) is rooted in different fields of research. They all share the vision to interweave the virtual world and the physical world.

In parallel, advances in microelectronics, microelectromechanical systems (MEMS), and wireless communications allowed for the miniaturization of networked computers as well as sensors and actuators to connect to the physical world. This technology opened new fields that are also considered an integral part of the IoT as depicted in Figure 1.1:

- Wireless Sensor Networks (WSNs) leverage low-power radios and multihop communication to cover large areas with small, inexpensive, autonomous sensor nodes. They enable real-time sensor readings of physical phenomena, e.g., for battlefield surveillance, environmental monitoring, or smart cities [99, 157, 162].

- Machine-to-machine (M2M), which is rooted in the classic field of telemetry, usually uses cellular networks to connect stationary sensors and mobile objects, such as cargo or car fleets, to a central IT system [129, 200]. Besides cellular networks, there are new long-range radio technologies that target machine-to-machine (M2M), for instance *LoRA*[1], *Sigfox*[2], and the white space spectrum special interest group *Weightless*[3].

- *Smart Objects* are the continuation of Marc Weiser's classic vision of ubiquitous computing. Everyday objects are endowed with processing and communication capabilities together with sensors and/or actuators. Through the connection with digital services, these objects become 'smart' and can provide human–computer interaction that is woven into our everyday lives [73, 191].

All four fields can already be found in the real world. However, most of them form so-called silo applications. These closed vertical systems only fulfill a special task and are hard to integrate with systems from other application domains.

The crucial leap toward a literal Internet of Things was made by adopting the Internet Protocol (IP) as the narrow waist to interconnect physical objects [169, 186]. In 2003, the IoT pioneers Adam Dunkels and Zach Shelby independently showed that native IP support is feasible for the resource-constrained devices used in wireless sensor networks (WSNs) and smart objects [44, 173]. With the increasing interest in low-power networks, the IETF chartered a working group[4] in 2006 to standardize an adaptation layer for transmitting IP packets over IEEE 802.15.4 [1], the most common low-power radio standard at the time. The resulting IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) specifications [92, 135, 171] are based on the Internet Protocol version 6 (IPv6), which has a modular design, and hence is better suited for adaptation than its predecessor Internet Protocol version 4 (IPv4). Furthermore, IPv6 provides an 128-bit address space that

[1] http://www.semtech.com/ (accessed on 12 Feb 2015)
[2] http://www.sigfox.com/ (accessed on 12 Feb 2015)
[3] http://www.weightless.org/ (accessed on 12 Feb 2015)
[4] http://tools.ietf.org/wg/6lowpan/ (accessed on 12 Feb 2015)

theoretically could address $6.67 \cdot 10^{23}$ devices in every square meter of the Earth's surface. The actual number is much smaller due to the logical structuring of IPv6 addresses. Yet still, the number is large enough to globally address every single computer connected to the Internet in the foreseeable future. The 6LoWPAN binding is also being extended to emerging radio technologies such as DECT ultra low energy (DECT ULE) and Bluetooth Low Energy (BLE).[5]

This IP-based IoT now enables the seamless integration of the physical world into the virtual world represented by our computer systems that are globally connected through the Internet. The use of IP also fosters the convergence of the early, isolated IoT systems mentioned above. Increasing connectivity and removing the divide between application domains will enable novel applications and business models, ultimately creating a new economy. Studies by research firms predict more than 50 billion connected devices and a total economic value add of 1.9 trillion dollars by the end of 2020 [3, 124, 125].

1.1 Motivation

To live up to its expectations, the IoT must tear down its vertical silo architecture and enable connectivity between all application domains. This requires seamless interoperability at the application layer. To this end, the WoT initiative [81, 197] aims at adapting well-known patterns from the World Wide Web. The Web is the de facto application layer of the Internet and its technology provides the basis for most of today's digital services. Its fundamental architecture, analyzed and formalized by Roy Fielding as the Representational State Transfer (REST) architectural style [71], is highly versatile and flexible. It mastered the transition from static document retrieval to Web 2.0 applications and today's social networks. Using the Hypertext Transfer Protocol (HTTP), applications can interoperate and easily combine several services from different providers to create services of higher value, so-called Web mashups. In a similar way, Web technology can enable physical mashups for the IoT, that is, combine services of different devices that can also belong to different application domains [80]. Browsing the Web has become part of our everyday lives and, moreover, many tech-savvy people with minimal training are able to create their own Web applications. This is the second reason why the WoT initiative aims at applying the well-known and proven patterns from the Web to the demanding IoT domain: improved usability to manage the complexity of hundreds of billions of connected devices across all application domains. The estimated figures for the IoT in 2020 will only hold when there are enough developers to implement the novel applications and business models, and bring them quickly to the market.

[5] http://tools.ietf.org/wg/6lo/ (accessed on 12 Feb 2015)

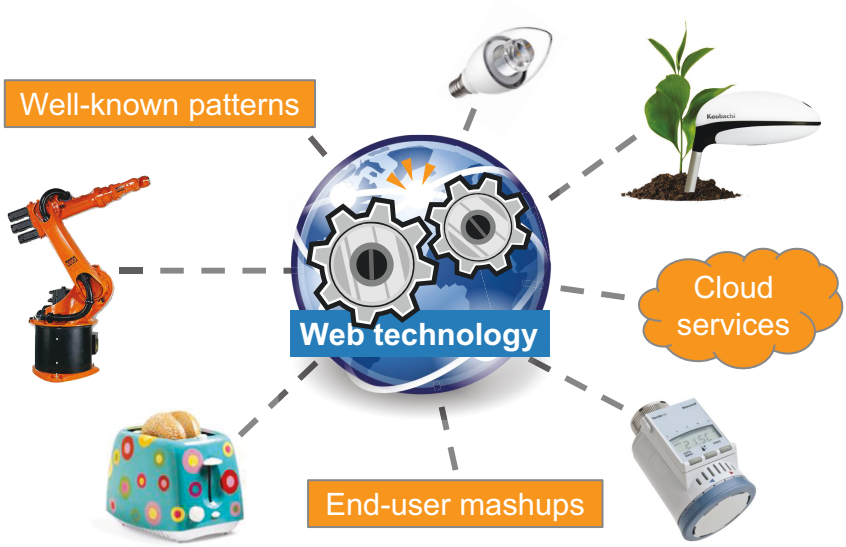


Figure 1.2: The WoT initiative promotes the adaption of well-known patterns from the World Wide Web to implement IoT applications, which shall connect different domains.

However, HTTP is a bad fit for the resource-constrained devices of Wireless Sensor Networks (WSNs), M2M, and smart objects, each representing a major building block of the IoT. On the one hand, the underlying Transmission Control Protocol (TCP) performs poorly in low-power wireless networks [9, 89, 90]. On the other hand, the verbosity of HTTP strains the memory buffers of devices and requires a considerable amount of bandwidth [40]. Most constrained implementations of HTTP hence only feature a minimal subset of the protocol that misses crucial REST features such as cache control or content negotiation. The benefits of Web technology need to apply end-to-end: Web-like application-layer interoperability must reach down to the device-level to allow for full convergence of the different application domains. Current WoT solutions require application-level gateways to integrate resource-constrained devices. If the application changes or a new device type is introduced, the gateway needs to be updated. This approach fails the end-to-end arguments of the Internet [20]. Furthermore, traditional programming models for networked embedded systems are insufficient for the IoT, as they were not designed for large-scale distributed applications; application-level gateways simply hide and shift the existing challenges of resource-constrained devices. To create a new economy, developers and users require improved usability, that is, models and tools that are adequate for the complexity of highly distributed applications. There are interesting approaches from the field of WSNs such as domain-specific languages or macro-programming. They are, however, only applicable for domain-specific applications and fail the goal of convergence.

Our thesis is that existing protocols and programming models do not effectually meet the needs of the IoT. We identify two key challenges for the vision to succeed: application-layer interoperability and improved usability for both developers and users. Both requirements can be met by an approach that amalgamates results from the field of Wireless Sensor Networks and the World Wide Web.

1.2 Research Goals and Contributions

The overall goal of this work is to provide application-layer interoperability and improved usability for the emerging Internet of Things (IoT). For this, we follow the WoT initiative and adapt Web technology to meet the requirements of the IP-based IoT, which is expected to consist of a huge number of tiny, inexpensive, resource-constrained devices. Based on this scenario, we structure our research questions into three aspects:

Scaling down *How can we scale Web technology down to constrained environments?* A large share of IoT devices will have limited processing power (i.e., microcontrollers), memory (i.e., about 100 KiB of ROM and 10 KiB of RAM), and energy (i.e., battery-powered or relying on energy harvesting), and they will be connected in low-power networks (i.e., low data rates and high message loss). Since HTTP is not a good fit for such constrained environments, the IoT requires a new, suitable Web protocol as well as proper tools. For this, we design concepts to close the gap between tiny resource-constrained devices and the World Wide Web.

Scaling up *How can we scale Web technology up to hundreds of billions of IoT devices?* With up to 212 billion estimated IoT devices connected by 2020, backend and support services must be able to handle vast concurrency factors. Given the new Web protocol for resource-constrained devices, we study high-performance HTTP server architectures to develop a scalable system architecture for IoT backend services. Its evaluation must reflect the new traffic patterns within the IoT, that is, short but numerous messages.

Improving usability *How does Web technology improve usability for developers&users?* Our hypothesis is that the proven patterns from the World Wide Web will also improve usability for developers and users of resource-constrained IoT systems. To elaborate on this hypothesis, we evaluate scripted IoT mashups at the device level and study Web browser integration of the new Web protocol.

Our contributions for scalable Web technology for the IoT across these three aspects can be summarized as follows:

1.2.1 Protocol Design and Evaluation

In the course of this dissertation, we contributed to the design and standardization of the Constrained Application Protocol (CoAP) within the IETF. Furthermore, we systematically evaluate the performance of the new Web protocol. Our studies show that an IoT protocol stack can be made power-efficient through a generic radio duty cycling (RDC) mechanism without changing the application-layer protocol. To the best of our knowledge, we are also the first to evaluate CoAP in unconstrained environments, that is, its performance for IoT cloud services. Based on our reference implementation, we evaluate how CoAP performs when interacting simultaneously with a huge number of devices.

1.2.2 System Architectures and Guidelines

We propose concrete system architectures and guidelines for correct implementation of CoAP and optimal utilization in IoT applications. For resource-constrained devices, we provide techniques to optimize the memory footprint, but also to provide a user-friendly application programming interface (API). For IoT cloud services, we present a scalable system architecture that outperforms existing CoAP solutions as well as high-performance HTTP servers.

1.2.3 Concepts and Tools

We present concepts and tools for Web-like programming models and interaction in the IoT. We show how the concept of Web mashups [80, 138] can be applied to IoT services directly hosted on resource-constrained devices. Our thin server architecture enables an application-agnostic device infrastructure that is able to serve multiple applications that can change over time. The Web-like programming model is supported by our Web browser prototype, which can be used to directly test, debug, and manage IoT devices and services.

1.2.4 Open Source Implementations

To support our thesis, we also provide open source implementations of our concepts. The *Erbium (Er) REST Engine* is optimized for resource-constrained devices and has become the default CoAP implementation for Contiki OS[6], a popular embedded operating system for the IoT. Erbium is available on GitHub in the Contiki repository[7].

[6] http://www.contiki-os.org/ (accessed on 12 Feb 2015)
[7] https://github.com/contiki-os/contiki (accessed on 12 Feb 2015)

Our Java-based *Californium (Cf) CoAP framework* targets IoT cloud services and less constrained devices (i.e., systems with megabytes of memory). The project was incubated at the Eclipse Foundation[8] and includes two sub-projects: the *Scandium (Sc)* DTLS 1.2 implementation, one of the first DTLS implementations available in Java, and the *Actinium (Ac)* runtime container for JavaScript-based IoT mashups. The Californium source code is available on GitHub in the Eclipse repositories[9].

Lastly, the *Copper (Cu) CoAP user-agent* is available as add-on for Firefox[10] and enables browser-based interaction with IoT devices, including linking, browsing, and bookmarking. Its source code is hosted on GitHub[11].

1.3 Outline

We first provide an introduction to the concepts and features of the Constrained Application Protocol (CoAP) in Chapter 2. This will define the important terminology for this dissertation and also give insights to the ideas and trade-offs behind the design decisions made by the IETF working group for Constrained RESTful Environments (CoRE).

After that, the structure of this document follows the three aspects of our research goals: Chapter 3 presents our concepts for scaling Web technology down to resource-constrained devices. We introduce our thin server architecture and discuss our prototypical evaluation of CoAP in constrained environments.

In Chapter 4, we present our approach to scale Web technology up to hundreds of billions of IoT devices. We propose a scalable architecture for CoAP-based IoT cloud services and evaluate the performance of the new Web protocol in the comparatively unconstrained service backend.

Our study on how Web technology improves the usability for developers and users is documented in Chapter 5. We propose a Web-like programming model for the IoT, study Web browser integration, and discuss the results of our user study.

Finally, we give a summary of our work and conclude our findings in Chapter 6. With this in mind, we discuss the open challenges and provide directions for interesting future work.

[8] http://www.eclipse.org/ (accessed on 12 Feb 2015)
[9] https://github.com/eclipse?query=californium (accessed on 12 Feb 2015)
[10] https://addons.mozilla.org/firefox/addon/copper-270430/ (accessed on 12 Feb 2015)
[11] https://github.com/mkovatsc/Copper (accessed on 12 Feb 2015)

Chapter 2

The Constrained Application Protocol

Moore's Law states that the number of components per integrated circuit at minimal cost doubles approximately every two years. This has led to powerful processors that integrate multiple cores and billions of transistors on a single chip. In the vision of the IoT, however, this technological advancement is not used to increase the computing power of devices, but primarily to decrease power consumption, miniaturize whole systems into a tiny chip, and in particular to minimize unit costs. This also means that most devices in the IoT will remain resource-constrained and will require lightweight protocols.

With the standardization of 6LoWPAN [92, 135, 171], the vision of an *Internet* of Things has become more concrete: heterogeneous low-cost computers that can be embedded in everyday objects are directly accessible through standard Internet protocols. However, because of the discussed problems of HTTP for resource-constrained devices and M2M applications (see Section 1.1), a common application layer has still been missing.

This gap is closed by the Constrained Application Protocol (CoAP), a new Web protocol that was accepted as Proposed Standard by the Internet Engineering Task Force (IETF) in July 2013 [172]. We are one of the main contributors to the design and technical specification of CoAP and its complementary extensions, which are maintained in separate documents [25, 84, 158, 170, 175]. We have been providing *running code*[1] for the IETF standardization, which is fundamental for the verification of the proposed protocol drafts. Being the first to integrate the base extensions in a single implementation and collect experience in evaluations, this dissertation contributes a significant part to the realization of the new Web protocol for resource-constrained devices.

[1] http://www.ietf.org/tao.html (accessed on 12 Feb 2015)

This chapter gives an overview of the protocol and explains the ideas and trade-offs behind the design decisions. First, we summarize the requirements in Section 2.1, which refines the general aspects given in the previous chapter. We continue with the protocol fundamentals (Section 2.2), which are part of the base specification. In Section 2.3, we highlight the protocol extensions available at the time of writing, which make the key difference for M2M applications. Next, we explain the discovery mechanisms used in CoAP (Section 2.4) and finally how the new protocol integrates with existing infrastructures (Section 2.5).

2.1 Constrained RESTful Environments

In 2010, a new working group for Constrained RESTful Environments (CoRE) was chartered at the IETF.[2] Based on the success of Web technology and the advantages of REST, the group has the goal to provide a framework for applications that embrace constrained IP networks as found in the IoT. These networks consist of resource-constrained devices, which can be classified according to their capabilities. RFC 7228 on terminology defines the following three classes [23]:

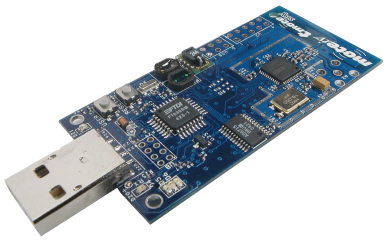


Figure 2.1: Tmote Sky

Class 0 devices are *not* capable of running an RFC-compliant IP stack *in a secure manner*. They usually run a proprietary, minimalistic protocol that is tailored to the application and platform. To connect to the Internet or IP-based IT systems in general, Class 0 devices require application-level gateways. These perform protocol translation and take over other responsibilities such as neighbor discovery and security. The *Tmote Sky* [154] shown in Figure 2.1 is such a highly resource-constrained platform: It only has 48 KiB of ROM and 10 KiB of RAM. It is able to implement an RFC-compliant IP stack together with an energy-saving MAC protocol and CoAP at the application layer. The program flash is, however, too small to also include the cryptographic libraries required for a full security handshake. Requiring either external support or extreme optimization that breaks interoperability makes the Tmote Sky a borderline Class 0 device. More obvious examples are proprietary temperature sensors that send their readings wirelessly to an indoor weather station or bedside alarm clock. Their memory sizes are usually in the order of hundreds of bytes only.

[2] https://datatracker.ietf.org/wg/core/history/ (accessed on 12 Feb 2015)

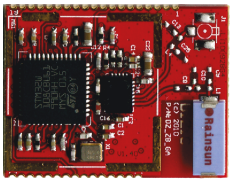


Figure 2.2: STM32W SoC

Class 1 devices are the most resource-constrained devices that can directly connect to the Internet with onboard security mechanisms. This requires about 100 KiB of ROM and about 10 KiB of RAM. They cannot employ a full protocol stack using HTTP over Transport Layer Security (TLS), though, due to limited memory size and processing power. Thus, they require lightweight protocols that have low memory footprints and parsing complexity. Platforms based on the ARM Cortex-M3 architecture, such as the *STM32W108* SoC shown in Figure 2.2, fall into this category. With 128 KiB of ROM and 16 KiB of RAM, this device has enough space for an optimized but complete network stack secured through DTLS [161]. Class 1 devices are in the focus of the Constrained RESTful Environments (CoRE) working group, the Constrained Application Protocol (CoAP), and this dissertation.

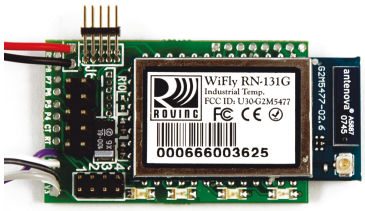


Figure 2.3: Roving RN-131

Class 2 devices almost show the characteristics of full-fledged Internet nodes like smartphones or notebooks. This becomes possible at about 250 KiB of ROM and about 50 KiB of RAM. Yet they can still benefit from lightweight and energy-efficient protocols to free resources for the application or reduce operational costs, for instance by having longer battery-replacement intervals. Figure 2.3 shows for instance a low-power Wi-Fi module with 2 MiB of ROM and 128 KiB of RAM. Its battery-lifetime depends on long hibernation phases and can thus benefit from low network overhead and short round-trip delay times (RTTs).

Other networking aspects are subject to constraints as well, such as the low achievable data rates and high packet loss. Class 1 devices mostly use low-power communications such as IEEE 802.15.4 or BLE. These have short link-layer frames[3] to reduce data corruption due to interference and consequently the number of lost frames. Hence, message sizes should be small to suffice the limited network buffers and minimize fragmentation.

All this renders HTTP over TCP and its heavyweight M2M solutions unsuitable for such environments. Thus, Web technology for the IoT calls for a new RESTful protocol that is compact, has low parsing complexity, can handle lossy communication links, and provides features for typical M2M applications.

[3] IEEE 802.15.4 provides up to 127 octets and BLE only 47 octets of data at the data link layer.

2.2 Protocol Fundamentals

To fulfill the requirements of constrained RESTful environments, the IETF working group designed the Constrained Application Protocol (CoAP), which goes beyond a mere compression of HTTP. Following the REST architectural style as defined by Roy Fielding [71], it is primarily based on patterns from the Web: a client/server interaction model between application endpoints, resources that are addressable by Uniform Resource Identifiers (URIs), stateless exchange of representations that decouple client and server, uniform interfaces with standardized Internet Media Types for wide interoperability, and caching and proxying to enable high scalability. CoAP uses the User Datagram Protocol (UDP) as transport, which offers better performance than TCP in low-power wireless communication with lossy links [9, 89, 90]. Primarily, UDP has less overhead, since it has a smaller header, requires less code, and omits connection setup and tear-down. Applications may require a higher quality of service than offered by UDP, though. Thus, CoAP implements a thin control layer below its request-response model as depicted in Figure 2.4.

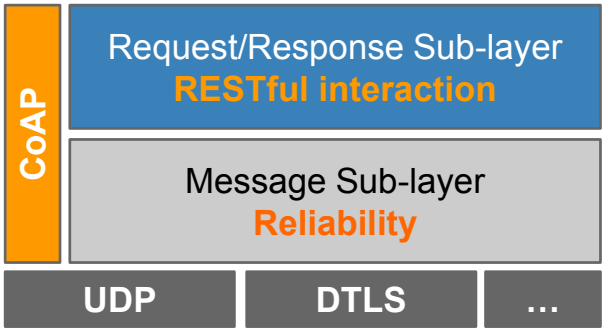


Figure 2.4: CoAP is organized in two sub-layers: the request-response sub-layer for RESTful interaction and the message sub-layer for deduplication and optional retransmissions.

2.2.1 Message Format

CoAP is a binary-encoded communication protocol, which allows for compact messages and low parsing complexity for microcontroller-based systems. Figure 2.5 shows the four-byte base header of CoAP, which can be followed by the variable-length token, multiple header options, and a payload carrying the representations mentioned above.

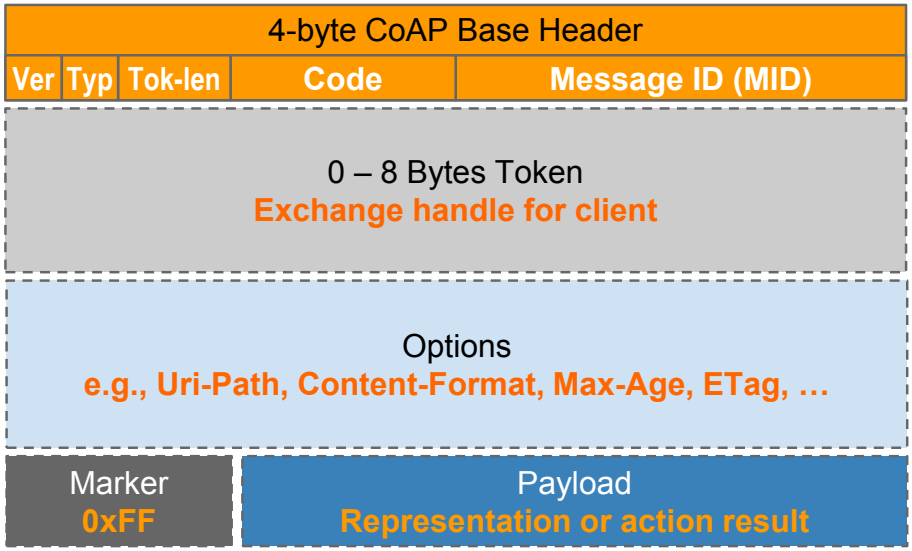


Figure 2.5: The CoAP message format has a 4-byte base header that can be followed by a token, options, and a payload. The base header contains two bits for versioning, two bits to encode the message type, four bits for the token length, one byte for the message code (a RESTful method or a response code), and 2 bytes for the message identifier (MID).

2.2.2 Messaging Sub-layer

The messaging sub-layer provides duplicate detection and optionally reliable transmission of messages. The latter is based on a simple stop-and-wait mechanism for retransmissions with truncated binary exponential backoff. For this, the sub-layer uses 16-bit message identifiers (MIDs) and four different message types.

Message Types

Confirmable (CON) messages provide reliable transmission. They are retransmitted until the receiver confirms their reception or they ultimately time out (see Figure 2.6). CoAP uses a random initial retransmission timeout (RTO) t_{init} with the protocol parameters $T_0 = 2\ s$ and $C_{rand} = 1.5$ to define the interval bounds:

$$\begin{aligned} T_0 \leq t_{init} &\leq T_0 \cdot C_{rand} \\ 2\ s \leq t_{init} &\leq 3\ s \end{aligned} \tag{2.1}$$

This random backoff avoids synchronization effects, e.g., when all nodes in a low-power lossy network (LLN) start to operate at the same time after a power outage. After each retransmission, the current timeout is doubled until the retransmission counter reaches $R_{max} = 4$. In total, a CON message will be sent up to five times when it is not acknowledged.

ACK messages are replies to CON messages and acknowledge the transmission using the same MID for correlation. Usually, an ACK piggy-backs a response to directly respond to a CON request. In case the generation of a response requires some time and is sent separately in a new transmission (e.g., due to a slow sensor or a long computation) or the CON message carries such a separate response, an empty ACK is used to close the transmission. Empty ACKs only consist of the 4-byte base header, have a code of zero, and no options nor payload.

Non-confirmable (NON) messages are used for best-effort delivery, for instance when the loss of a single message is acceptable because it is triggered in a regular interval or a retransmission would only interfere with a succeeding message. NON messages should be answered with another NON message. However, mixing NON and CON–ACK pairs is also possible and standard-compliant implementations must be prepared for this. In both cases, the reply message must use a new, sender-generated MID to enable reliable transmission. The correlation of requests and responses is done in the request-response sub-layer.

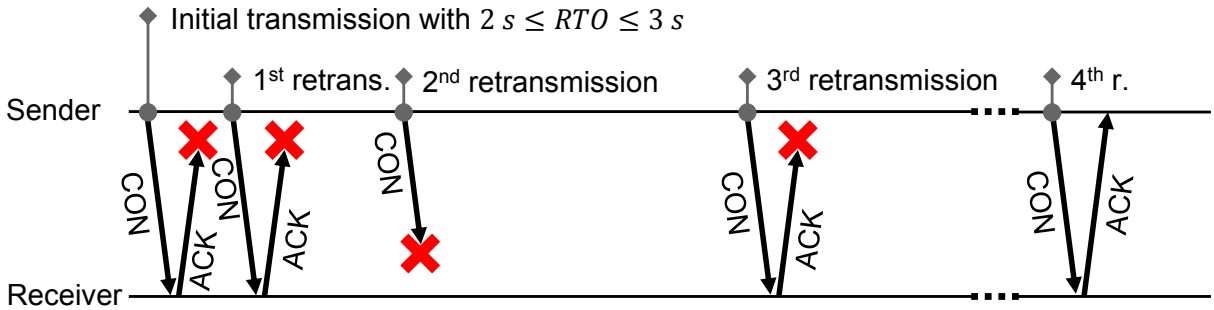


Figure 2.6: Reliable transmission: CON messages must be confirmed with an ACK message. If no acknowledgement is received, e.g., due to loss of the CON or the corresponding ACK, CONs are retransmitted up to four times. Thereby, the initial retransmission timeout, which is randomly chosen between two and three seconds, is doubled after each send event.

Reset (RST) messages are replies to CON and NON messages to indicate message processing errors due to missing context on the receiver side. This can happen when a node rebooted and cannot match the message to its open exchanges or is otherwise unable to accept the message. Like ACKs, the replies must use the same MID as the received message and close reliable transmissions, but inform about failure. Sending a RST is optional for received NONs, since they can be silently dropped. Rejecting NONs can help to avoid repeatedly sending effectless messages, though, for instance unwanted push notifications.

Deduplication

Duplicate messages are usually caused when ACK messages are lost and a CoAP endpoint keeps retransmitting CON messages. Yet unreliable transports such as UDP are also prone to message duplication by the network itself. In CoAP, both are solved through filtering based on the 16-bit MID: Every active CON and NON message must use a MID that is unique within its source endpoint, which is identified by IP address and UDP port number. Messages are no longer active when there is no copy residing in the network anymore. This *message lifetime* depends on the transmission parameters and message type. For CONs, it is derived as follows:

First, the MID is used at least as long as it is transmitted. This maximum transmission time T_{tx} results from the exponential backoff and number of restransmissions R_{max}:

$$\begin{aligned} T_{tx} &= T_0 \cdot C_{rand} + \sum_{i=1}^{R_{max}-1} T_0 \cdot C_{rand} \cdot 2^i \\ T_{tx} &= \sum_{i=0}^{R_{max}-1} T_0 \cdot C_{rand} \cdot 2^i \\ &= T_0 \cdot C_{rand} \cdot (2^{R_{max}} - 1) \\ &= 2\ s \cdot 1.5 \cdot 15 = 45\ s \end{aligned} \tag{2.2}$$

In addition, each message requires time to travel through the network before it arrives at the receiver. In the best case, this is close to zero. The worst case is modeled with the maximum network latency T_{lat}. This upper bound is chosen in the order of the Maximum Segment Lifetime of TCP [156], but simplified to $T_{lat} = 100\ s$. The worst case arrival time for a CON message occurs when the first four transmissions are lost and is estimated with $T_{tx} + T_{lat} = 145\ s$.

The upper bound for the recipient to process the message and generate a reply, which can be an empty ACK to signal a later separate response (see Figure 2.9), must be within the initial RTO T_0. This bound is thus defined to $T_{proc} = 2\ s$. In the worst case, this reply will travel back through the network with T_{lat} before it eventually arrives at the sender of the CON message. The whole process is depicted in Figure 2.7 and results in an overall exchange lifetime τ_{CON} for CONs of:

$$\begin{aligned} \tau_{CON} &= T_{tx} + T_{lat} + T_{proc} + T_{lat} \\ &= 45\ s + 100\ s + 2\ s + 100\ s \\ &= 247\ s \end{aligned} \tag{2.3}$$

For NON messages, the MID is only used for one direction, since the reply needs to use a new NON or CON message. The same NON is, however, also allowed to be sent multiple times to increase robustness. For simplification, the upper bound for this sending time span is estimated with T_{tx} as well. This results in a lifetime τ_{NON} for NONs of:

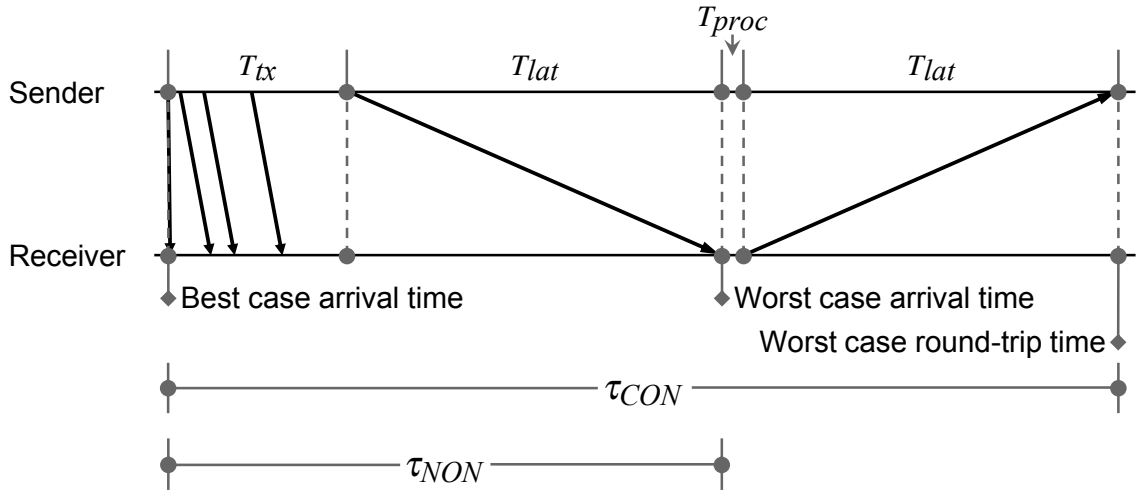


Figure 2.7: The MID-based deduplication mechanism requires an estimation of the message lifetimes: In the best case, the network latency is close to zero and the first transmission successfully arrives at the receiver virtually at the same time it was sent. In the worst case, the first four transmissions fail and last (re)transmission experiences the maximum network latency. The lifetime of CON messages (τ_{CON}) is derived from the worst case RTT, where the receiver adds a maximum processing delay and the ACK reply also experiences the maximum network latency. The lifetime of NON messages (τ_{NON}) is derived from the worst case arrival time, since replies to NON are sent in a new NON or sometimes CON transmission with independent MID.

$$\begin{aligned}\tau_{NON} &= T_{tx} + T_{lat} \\ &= 45\ s + 100\ s \\ &= 145\ s\end{aligned} \tag{2.4}$$

These estimations describe the delays an implementation needs to take into account to enable safe deduplication. In an optimal implementation, the receiver must keep its filter entries for the worst case arrival time, which is $T_{tx} + T_{lat} = 145\ s$. Since the receiver cannot decide which transmission it received, senders cannot simply reuse the MID after its corresponding message lifetime. In the worst case for CON messages, the receiver only sees the last retransmission after $T_{tx} + T_{lat}$ and will filter it for the same amount of time. This corresponds to $2 \cdot (T_{tx} + T_{lat}) = 290\ s$, which is longer than τ_{CON}. Since implementations might choose to keep deduplication entries for even longer (e.g., using the full message lifetime or a time span that fits better to their internal timer implementation), senders should use a conservative timespan before reusing a MID.

2.2.3 Request-Response Sub-layer

The request-response sub-layer implements the REST architectural style [71]. Thus, CoAP has common grounds with HTTP 1.1 [69], but also some distinctions, since CoAP aims at constrained environments and primarily M2M scenarios. The code field (see Figure 2.5) either carries a method code, making the message a request, or a status code, turning a message into a response. CoAP defines four RESTful verbs to interact with resources: GET, PUT, POST, and DELETE. They have the same semantics as their HTTP counterparts, which enables a stateless, transparent mapping. Also the response codes are defined with references to HTTP 1.1. There are, however, a few differences to make the codes more meaningful for caching intermediaries and M2M communication in general, e.g., explicit `2.02 Deleted` and `2.04 Changed` codes instead of the `204 No Content` status of HTTP. The notation for CoAP status codes uses a period to separate the class from the detailed code instead of the implicit hundreds digit in HTTP. This is because CoAP codes are represented internally with 8-bit numbers. The three most significant bits of the message code field carry the class and the five least significant bits the detail. Accordingly, the integer encoding can be calculated as $CODE = CLASS \cdot 32 + DETAIL$. In addition to the code and possible action results (i.e., representations that do not belong to a resource, but result from an action), error responses can have a *diagnostic payload*, which is not a representation but a short, human-readable message explaining the reason for the failure to software engineers. Table 2.1 summarizes the methods and response codes currently defined by CoAP [172].

Code	Description	Comment	HTTP
0.xx	**Methods**		
1	GET	safe, idempotent	GET
2	POST	—	POST
3	PUT	idempotent	PUT
4	DELETE	idempotent	DELETE
2.xx	**Success**		
2.01	Created	in response to POST or PUT	201
2.02	Deleted	in response to DELETE or POST	204/200
2.03	Valid	in response to GET with ETag	304
2.04	Changed	in response to POST or PUT	204/200
2.05	Content	in response to GET	200
4.xx	**Client Error**		
4.00	Bad Request		
4.01	Unauthorized	no WWW-Authenticate header (thus no HTTP 401 Unauthorized mapping)	400
4.02	Bad Option	for unrecognized or malformed options	400
4.03	Forbidden	for general denial independent from authentication	403
4.04	Not Found		
4.05	Method Not Allowed	no Allow header in CoAP	400
4.06	Not Acceptable		406
4.12	Precondition Failed	based on preconditions defined through options	412
4.13	Request Entity Too Large	maximum size can be included in a response option	413
4.15	Unsupported Content-Format	unsupported request payload	415
5.xx	**Server Error**		
5.00	Internal Server Error		500
5.01	Not Implemented		501
5.02	Bad Gateway		502
5.03	Service Unavailable		503
5.04	Gateway Timeout		504
5.05	Proxying Not Supported		502

Table 2.1: The response codes defined by RFC 7252 and a possible mapping to HTTP.

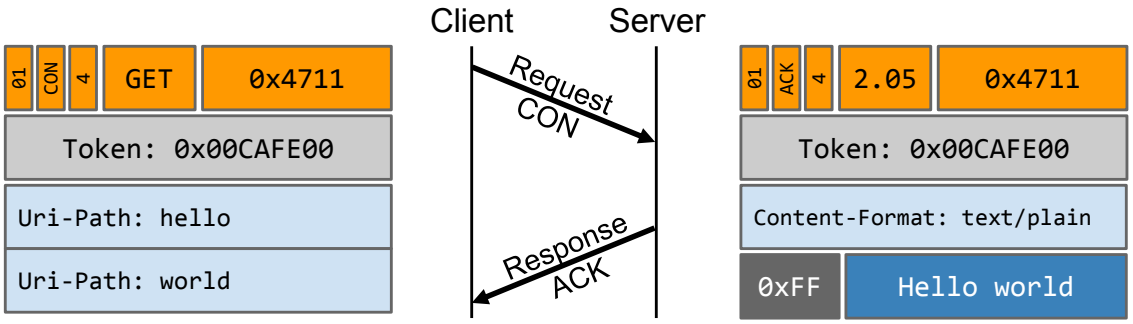


Figure 2.8: Usually, a request-response exchange requires two messages, as the ACK replying to a CON request can piggyback the response. Note that matching a response to a request is done through the token, not the MID.

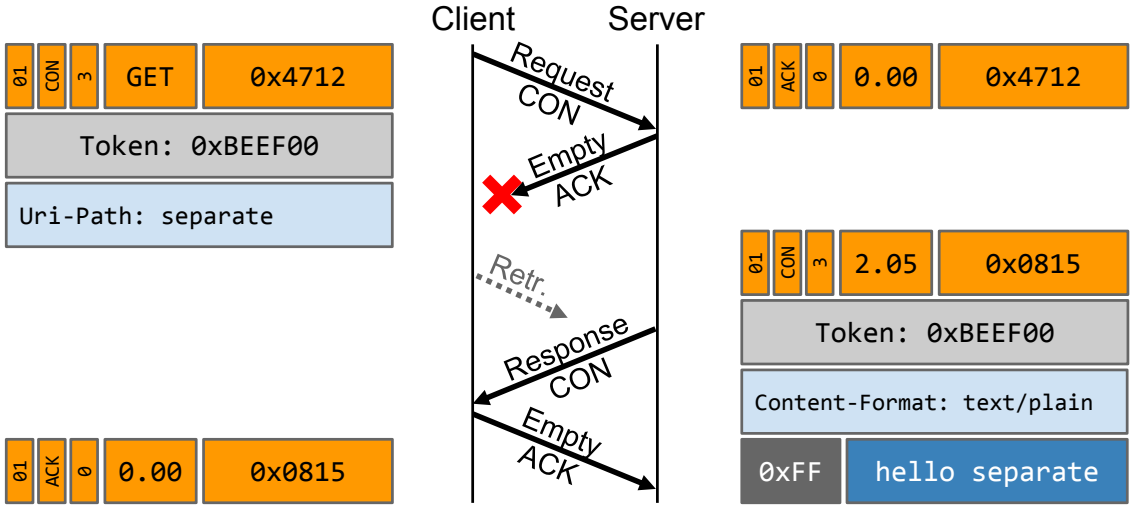


Figure 2.9: When the creation of a response takes some time, e.g., longer than the initial RTO, the request is acknowledged with an empty ACK and the response is sent as a separate transmission using a CON message as well. NON requests always elicit a separate response where the response has its own MID and must be correlated through the token. When the client receives the response for a CON request that has not been acknowledged yet, it may stop its retransmissions, as the response is an implicit acknowledgement for the request.

CoAP requests are sent either as CON or as NON messages depending on the desired reliability. To match the response message, which can be either, clients define an opaque token of up to eight bytes (see Figure 2.5), which must be mirrored by the server. For this purpose, it must also be unique for each open request or at least per destination if the server address is also used for matching. The token can also have a length of zero, the so-called empty token, which minimizes the message size. When endpoints are accessible over the Internet without further security precautions, however, clients should always generate a randomized token of four bytes or more to avoid response spoofing by off-path attackers.[4] Note that the token is independent from the MID, which is only used at the messaging sub-layer. In many cases, an ACK confirming a CON request can directly piggy-back the response (see Figure 2.8). Separate responses carried in a CON or NON message, however, have a different MID, which is defined by the server and cannot be used for response correlation. Table 2.2 summarizes the use of CoAP messages for requests and responses.

	CON	NON	ACK	RST
Request	✓	✓		
Response	✓	✓	✓	
Empty	Ping		✓	✓

Table 2.2: Usage of message types: An empty CON can be used to elicit a RST, the so-called CoAP ping.

2.2.4 Options

Similar to HTTP header fields, CoAP requests and responses can have options to specify additional but optional message semantics. To minimize message overhead, CoAP uses registered option numbers together with a binary type-length-value encoding. When an option has to specify multiple values, it is simply packed into multiple options with the same number. The CoAP specification [172] defines an initial Option Number Registry for basic options such as `Accept` or `ETag`. New option numbers can be registered through the Internet Assigned Numbers Authority (IANA). The registry[5] for this is split into three ranges as depicted in Table 2.3, which also shows a fourth range for the experimental options that cannot be registered.

[4] The NoSec mode should only be used in closed (private) networks that provide lower-layer security mechanisms such as IPSec [103] or Layer 2 encryption.

[5] http://www.iana.org/assignments/core-parameters/core-parameters.xhtml (accessed on 12 Feb 2015)

Option Number	Policy
0 – 255	IETF Review or IESG Approval Required
256 – 2047	Specification Required
2048 – 64999	Expert Review Required
65000 – 65535	Experimental Use

Table 2.3: The policy to register an option number depends on its value: lower numbers use less bytes, and hence are reserved for options defined in complementary IETF specifications. Higher numbers can be registered by other specifications and can also be vendor-specific.

A difference to HTTP is that the Request-URI is encoded in multiple options instead of the base header. This lowers the parsing complexity, as URIs are already dissected into `Uri-Host`, `Uri-Port`, and `Uri-Path` and `Uri-Query` segments when transmitted, each encoded in a separate option. With a pre-parsed URI, no percent encoding is required in CoAP, either. Resolving the target Web resource becomes very simple, as servers can directly match the segments to their resource tree. `Uri-Path` is a good example for repeatable options: each segment is encoded in a new option but with the same option number. The other basic options are comparable to HTTP and handle content negotiation (`Accept` and `Content-Format`), caching (`Max-Age` and `ETag`), conditional requests (`If-(None)-Match`), and linking (`Location-Uri` and `Location-Query`).

The encoding mechanism for options is optimized for size and enforces correct ordering. This is done through a delta encoding using nibbles (i.e., half octets) as explained in Figure 2.10 and Figure 2.11. Originally, a 4-bit option counter was used in the base header. Experience throughout the drafting phase showed, however, that a maximum of 15 options does not suffice. In particular, the limitation to the URI path depth (each segment is encoded in one option) would have been unacceptable for several industry application profiles. Hence, an end-of-options marker was introduced, which ultimately became the payload marker (i.e., it is only included if a payload follows the options). The freed up space was used to move the token from a header option to the base header, which is cleaner since the token is a fundamental mechanism for the message-based protocol and hence non-optional.

Option Numbers (i.e., the integers calculated from the Option Delta) also have properties encoded in the five least significant bits. This helps CoAP endpoints and intermediaries to handle options they do not know. The least significant bit decides whether an option is *critical* (odd) or *elective* (even). Critical options must be supported by the endpoint to correctly process the request or response. When in a request, servers must reject unsupported and/or unknown critical options with a `4.02 Bad Option` response. When

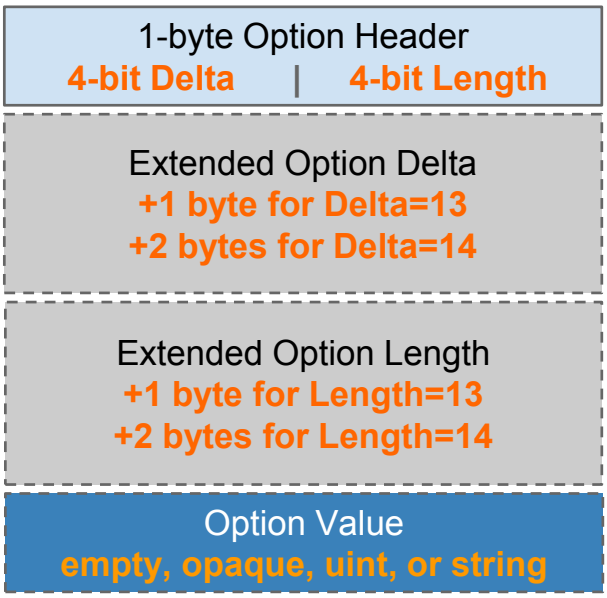


Figure 2.10: Options have a 1-byte header that contains a 4-bit Option Delta and a 4-bit Option Length. The delta encoding allows for a compact representation and a strict ordering of the options, so that Uri-Path segements, for instance, are always carried after the Uri-Host and before the Uri-Query to facilitate decoding. Repeatable options use a delta of zero. The length can also be zero to indicate an empty option, which can also represent an integer value of zero. Since the range of four bits is too small to encode large option number deltas or long option lengths, the values 13 and 14 are used to indicate one or two additional bytes to encode the value. With one additional byte, the range is 13–268, and with two bytes 269–65,804 (although the largest Option Number and maximum UDP size is 65,535).

it occurs in a response, clients either send a RST, a must for CON separate responses, or reject it by silently ignoring ACK and NON responses. Unsupported elective options are silently removed before processing the message. This procedure applies to clients and origin servers.

The other four bits tell proxies how to handle unknown options. When they are marked as safe to forward, actually encoded as the *Unsafe* bit set to zero, a proxy can forward such options to the origin server or back to the client without understanding them, even when they are critical. Together with the remaining bits, a proxy can decide whether the unknown option is part of the cache key. That is, a new cache entry is required, when one of the cache key options differs in value or presence. Also, only options that are safe to forward can be part of the cache key, as all others are either rejected or silently removed by the proxy. Being irrelevant for the cache key is rare, though. Thus, only when all three

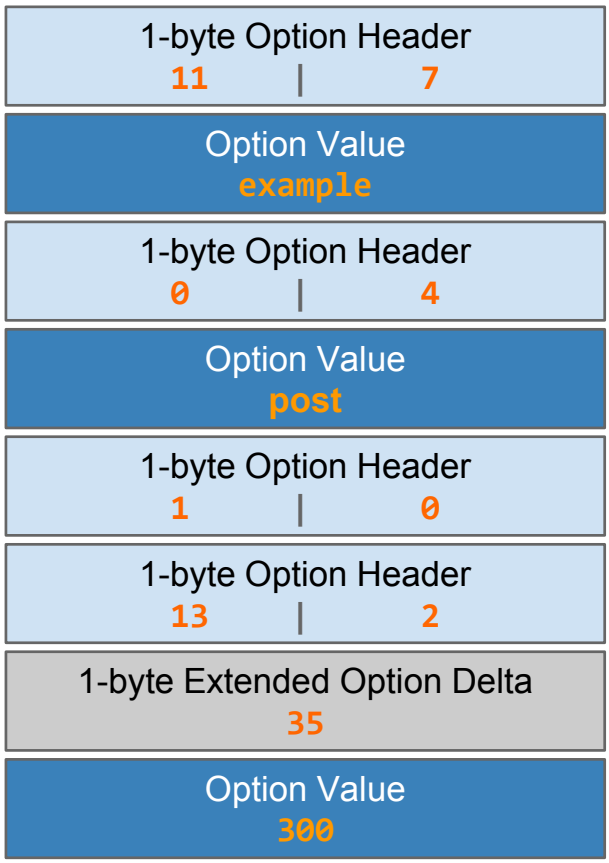
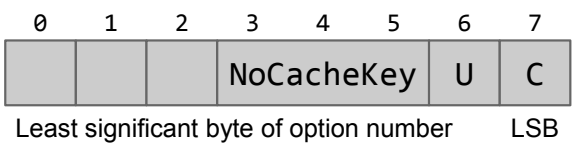


Figure 2.11: This example shows the encoded options for a POST request to a resource at `/example/post` that carries a `text/plain` representation of 300 bytes. When decoding, the Option Delta is added to a counter that holds the value of the current Option Number. It is initialized with zero, so adding the first delta results in Option Number 11 (`Uri-Path`). The second option has a delta of zero, and hence is a repeated `Uri-Path` option. The third option has a delta of one resulting in Option Number 12 (`Content-Format`). The zero-length indicates an uint value of zero, thus there is no option value. The last option is `Size1` with Option Number 60. It requires one additional byte to encode an Option Delta of 48(=13+35).

NoCacheKey bits are set to one and the option is safe to forward, it can be ignored for the cache entry. So theoretically only $P_{NoCacheKey} = 0.5^4 = 1/16 = 6.25\%$ of all possible options can have this property ($N_{NoCacheKey} = 4,096$). Figure 2.12 depicts the encoding of the option properties.



(a) Option metadata

```
Critical = (onum & 1);
UnSafe = (onum & 2);
NoCacheKey =
  ((onum & 0x1e)==0x1c);
```

(b) Metadata bitmasks

Figure 2.12: The least significant byte of the Option Number carries metadata in its least significant bits (note the big-endian notation for network byte-order).

2.2.5 Payload and Content

REST means exchanging resource state in form of representations of the Web resource. They are carried as message payload in most responses as well as POST and PUT requests. CoAP combines the `Content-Type` and `Content-Encoding` options of HTTP in a single `Content-Format` option to specify the enclosed Internet Media Type. The format is encoded as number, which is managed in an IANA registry using a variety of policies comparable to the option numbers. A content format entry can also include parameters, which would be appended with a semicolon in HTTP. An example is the `charset` parameter for `text/plain`: since CoAP globally uses the UTF-8 encoding, `Content-Format` value zero identifies the Internet Media Type `text/plain; charset=utf-8`.

Except for the mentioned diagnostic payload, CoAP applications should always specify the content format. This is not enforced by the specification, so closed systems can infer the format from context and hence minimize message sizes. Note that there is no default value when the `Content-Format` option is absent. The content format entries of the current IANA registry are given in Table 2.4. They provide the most generic Internet Media Types for M2M applications. Further entries should define specific formats that already provide the semantic information about how to handle the representations in order to use the RESTful Web service. This is an integral part of the Hypermedia as the Engine of Application State (HATEOAS) concept of REST.

A late change in the design of CoAP was to constrain and optimize the RESTful content negotiation for M2M communication. From HTTP, we know two main patterns, proactive and reactive negotiation [70]. The first is server-driven and uses a 'best guess' algorithm that selects the preferred representation based on the negotiation header fields (`Accept(-*)`), but also implicit client properties such as the network address. This pattern brings a few disadvantages, however, in particular inefficient selection and high complexity at the server. The second, reactive negotiation, performs selection on the client side based on a list of alternative resources provided through hypermedia links by the server. The most prominent example is choosing the language of a Web site through a list of flag

Identifier	Internet Media Type (and Parameter)	Encoding
0 – 255	**Expert Review Required**	
0	`text/plain;charset=utf-8`	–
40	`application/link-format`	–
41	`application/xml`	–
42	`application/octet-stream`	–
47	`application/exi`	–
50	`application/json`	–
60	`application/cbor`	–
256 – 9999	**IETF Review or IESG Approval Required**	
10000 – 64999	**First Come First Served**	
65000 – 65535	**Free Experimental Use**	

Table 2.4: The number assignment for CoAP content formats follows a policy similar to the one for the option numbers.

image links. Other, potentially complementary patterns even employ code-on-demand or intermediaries to select the right content. The negotiation in M2M communication is much more restricted, though. Language preferences are not required and constrained servers can only serialize a very limited number of Internet media Types. Most likely, a constrained client is only interested in a single representation it can process. Thus, CoAP uses a non-repeatable, critical `Accept` option that allows the selection of a specific representations and leads to an error response (`4.06 Not Acceptable` or `4.02 Bad Option` if unknown) in case it is not supported by the server. It is a combination of client-driven negotiation with the header option mechanism of proactive negotiation. This strengthens the resource abstraction because a single URI can be used for multiple representations of the same information (cf. different languages are usually provided as different resources). Moreover, it simplifies the implementation of proxies where permutations of a repeatable `Accept` option would result in different cache keys. Caching would become either less efficient or more complex because of duplicate detection. A minor advantage is also that CoAP implementations become smaller, since `Accept` had been the only repeatable integer option before the change and required its own code path.

One might argue that a single `Accept` value results in an inefficient trial-and-error method to find a supported representation of the resource. However, the discovery mechanism of CoAP is based on out-of-band information, which already provides a list of supported content formats (see Section 2.4.1). These attributes can also be added to links in hypermedia, so that clients know the available formats a priori.

2.2.6 Security

The security model of CoAP is similar to traditional Web services: Transport Layer Security (TLS). Due to the resource constraints and UDP binding, CoAP employs a UDP-based variant called Datagram Transport Layer Security (DTLS), which is defined as delta to TLS 1.2 [42, 161]. It provides the same flexibility with a variety of cipher suites, which define the set of cryptographic algorithms used.

Besides the NoSec mode, which elides DTLS completely, it is mandatory for CoAP endpoints to implement the RawPublicKey mode. Here, each device has an asymmetric key pair and a connected identity. For authentication, devices use an out-of-band mechanism based on the public key instead of full X.509 certificates, thus *raw public key* [199]. Optionally, CoAP nodes can also use the full X.509 Certificate mode or the more lightweight PreSharedKey mode. The latter uses a well-known AES block cipher with symmetric keys (`TLS_PSK_WITH_AES_128_CCM_8`). For asymmetric public-key cryptography, resource-constrained devices usually make use of elliptic curve cryptography (ECC), such as the `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` cipher suite, which is mandatory for CoAP and provides Diffie-Hellman key establishment with perfect forward secrecy.

2.2.7 Group Communication

Through UDP, CoAP is able to use IP multicast for group communication. It requires a corresponding routing or forwarding protocol to be active on routers such as the Multicast Protocol for Low power and Lossy Networks (MPL) [91] or Protocol Independent Multicast – Sparse Mode (PIM-SM) [66] for less constrained networks. Groups can then be addressed through URIs that map to a multicast address, e.g., `coap://all.floor5.example.com/light`. Requests to a multicast address must use NON messages and the receiving servers are required to randomly delay their response for a time called Leisure to avoid congestion. The default is five seconds, but the Leisure $\tau_{Leisure}$ can also be adapted to the actual group size G, if message size S and target data rate R can be estimated:

$$\tau_{Leisure} = S \cdot G / R \tag{2.5}$$

A receiving server can also decide to suppress the response, for instance when the payload is empty (e.g., due to `Uri-Query` filtering; Section 2.4.1) or when a `2.04 Changed` is the expected outcome. A further restriction for requests is to use idempotent methods (GET, PUT, and DELETE) only, as the client cannot readily decide which group members did receive the request.

The group communication document [158] gives additional guidance and constraints for RESTful group interaction. It also defines a RESTful interface to manage group membership using JSON. All group manipulation requests are unicast and must use a secured DTLS session. Multicast requests themselves currently must operate in NoSec mode, as CoAP itself does not specify secure, nor reliable group communication. Nonetheless, a secure group communication mechanism is currently planned by multiple parties within the IETF.

2.3 Protocol Extensions

CoAP is designed to be highly modular, so that resource-constrained application endpoints only need to implement the features they actually require. Around the base protocol specification, the CoRE working group is defining several extensions to provide a complete framework for RESTful IoT applications and to deal with the particular properties of constrained environments. In the following, we provide an overview over the most relevant specifications and drafts that are available at the time of writing.

2.3.1 Observing Resources

A key feature for the IoT is observing resources [84]. The observe extension enables efficient server push notifications based on the observer pattern [72]. It is designed as an optional feature on top of GET with an elective `Observe` option that is set to zero by the client. If a server does not support it, this simply falls back to answering a normal GET request and clients can revert to polling. If the server supports this feature, it will respond with this option, which turns the response into a notification. The server promises to keep the interested client on its list of observers as long as possible and will push new representations whenever the observed resource changes. This extends the request-response pattern to a request/multiple-response pattern, where all notifications are correlated as usual through the token. CoAP notifications also make use of cache control, that is, they have a valid lifetime defined by the `Max-Age` option and are cacheable. Usually the server sends a new representation before `Max-Age` expires. When a representation becomes stale, the client assumes that it was dropped by the server (e.g., because of a reboot) and can re-register by sending another observe request using the same token. Also the options must be identical to the original observe registration, so that the request matches the cache key in case intermediaries are involved.

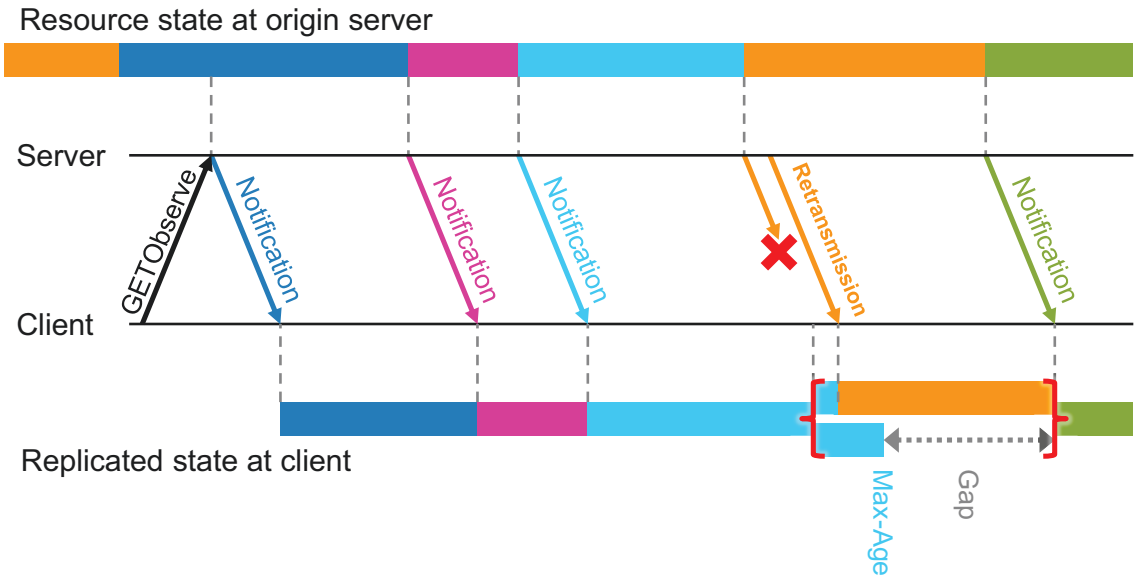


Figure 2.13: CoAP observe synchronizes the local state with the resource state at the origin server by sending push notifications. In case a notification is lost, the client can use its currently cached version until it expires. In the worst case, this might lead to gaps where the client would be forced to request the current state from the server, thereby usually re-registering its interest in observe notifications. Usually, NON notifications are used in combination with frequent updates, though, so that there is a low probability for gaps. Infrequent notifications usually use CON messages, which are retransmitted to ensure the synchronization of the replicated state at the client.

In case the client is no longer interested, there are two possibilities to end an observe relationship:

- **Re-active cancellation:** Observers can simply remove the local relationship states, which leads to a 'garbage collection' at the server: When the next notification arrives, the client cannot match the token and will reject it. Since every once in a while the server must use a CON notification to detect orphans, it will eventually receive a RST that tells the server to remove the client from its list of observers. The same happens when a CON transmission times out, usually caused by a client that shut down (orphan).

- **Pro-active cancellation:** Some applications require a more timely cancellation to save resources. In this case, clients can send a cancellation request by setting the `Observe` option to one and using the token associated with the relationship.

Originally, a GET request without `Observe` option was used to cancel a subscription. This led, however, to accidental cancellations by other parts of the application logic that

happen to use the same CoAP client instance. While the re-active cancellation would work in principle, it can waste lots of resources in a constrained environment. NON notifications do not expect a reply and incoming RSTs might be ignored because the implementation does not store the MIDs (e.g., to optimize memory usage). The maximum interval to check for orphans by introducing a CON notification for these cases is 24 hours. This may cause many useless packets draining energy in all forwarding LLN nodes or block scarce observer slots at the constrained server. Thus, the pro-active cancellation was added in version 13 of the observe draft specification.

An important concept behind observing resources is eventual consistency [84]: Unlike publish/subscribe, where the goal is to propagate every event, observe guarantees that eventually all registered observers will have a current representation of the latest resource state. This means that intermediate representations might be skipped when the changes occur too frequently for propagation over a constrained network. Nonetheless, CoAP can be used for eventing when following a correct RESTful design: Events must be modeled as state. A resource representing a doorbell, for instance, can be modeled as timestamp of the last event or a counter for how often it was pressed. Note that the observe sequence number carried in the `Observe` option of notifications cannot be used for the latter. It may start at any value and may increase with arbitrary steps that fulfill the clock condition: it may not increase by more than 2^{23} within less than 256 seconds.

2.3.2 Blockwise Transfers

6LoWPAN already provides a fragmentation mechanism. However, the maximum transmission unit (MTU) is only 1280 bytes. When resource representations become larger than the MTU, CoAP can still split the payload into multiple chunks using blockwise transfers [25]. This enables, for instance, firmware updates without resorting to an alternate protocol to transfer large data to or from a device. Blockwise transfers are also useful for messages smaller than 1280 bytes. They enable the incremental generation of large representations on the fly, which is beneficial for devices that can only process a limited amount of data at a time, which is often the case due to small buffer sizes. The most common use for blockwise transfers is resource discovery, as the CoRE Link Format [168] listing all available resources can easily grow to the order of kilobytes (see Section 2.4.1).

The extension defines four new options: The `Block1` option is used for the payload of the first part of a request-response exchange, that is, for PUT and POST requests. `Block2` is used for the second part, that is, the response payload. Besides the block size, the options specify the number of the current block and whether more blocks are following. Usually, the recipient is interested in the whole representation. Thus, it will request the

next block until this bit indicates no more blocks. As the overall size of the representation might exceed the memory of a receiving device, the extension also defines `Size1` and `Size2` options to give a current estimate of the overall size in advance. The actual size may differ because the exact number of bytes sometimes cannot be known for on-the-fly generation, for instance, when serializing numeric sensor values to JSON.

To allow for an efficient coding of the options, blockwise transfers define seven exponentially growing block sizes from 16 to 1024 bytes. This is not optimal for IEEE 802.15.4 frame sizes, as it leads to internal fragmentation, that is, frames that only carry a few bytes payload. Our evaluation in Chapter 3 shows, however, that the actual number of transmitted frames depends more on the MAC layer than on the optimal utilization of frame sizes and that RDC has a much higher impact on actual energy savings [107]. Furthermore, the 6LoWPAN binding is currently extended to other technologies such as BLE or DECT ULE, which have different frame sizes again.

When blockwise transfers are combined with observing resources, only the first block is sent as notification. The remaining blocks need to be fetched by the observers using normal GET requests. This significantly reduces the management overhead at the server, since the clients have to coordinate the transmission of all their blocks. This also lowers the requirements on congestion control on the server side.

2.3.3 Advanced Congestion Control

Being an Internet protocol, CoAP must adhere to congestion control, primarily to keep the backbone network stable, but also to avoid overrunning constrained networks and endpoints with too many messages. Thus, the base specification uses conservative parameter values for the number of open requests, the retransmission timers, and the overall message rate. More powerful CoAP nodes, however, can use metrics to optimize these parameters to achieve a better quality of service.

One proposal for a more sophisticated mechanism is CoAP Congestion Control Advanced (CoCoA) [22]. Its key idea is to use RTT measurements to adapt the RTOs, while keeping the focus on constrained node networks to avoid the re-invention of TCP. In particular, CoCoA uses to RTO estimators: A strong estimator that uses Karn's algorithm [102] and guidelines for TCP [151], as it only takes measurements when no retransmission occurs, and a weak estimator that takes RTTs measurements when a message was retransmitted. This estimator is less reliable, yet it allows to gather information in lossy environments to continuously update the overall RTO estimator. This initial RTO is then combined with an exponential backoff that uses a factor different from two (the base specification doubles the backoffs after each retransmission). A higher value is used when

the initial RTO is low, since a timeout is more likely caused by congestion than by message loss when RTTs are generally low. When the initial RTO is already high, a lower factor value is used to limit the transmission time span. The RTO is further truncated when it exceeds 60 seconds.

The RTT measurements also allow a dynamic adaptation for NONs. By default, the maximum rate is limited to 1 B/s. With CoCoA, this is replaced by a message rate of $\frac{1}{RTO}$ msg/s. All in all, this extension can increase the throughput of CoAP by 19–112% [16].

2.3.4 Alternative Transports

Although UDP is the primary transport for CoAP, the protocol was designed to run over alternatives as well [175]. For instance, there is a proposal to add CoAP bindings for the Short Message Service (SMS) [13]. Many IoT devices are connected through cellular networks that do not have IP connectivity to lower implementation costs, due to limited coverage, or a temporarily disconnect to conserve energy. All resources are still addressable through URIs such as `coap+sms://+123456789/container/bananas/temp`[6].

SMS comes with its own reliability mechanism and has support for delay-tolerant delivery. This is interesting for logistics, for instance, when freight containers leave cellular coverage at sea or in the air. CoAP messages can occupy up to 140 octets over SMS. SMS can also concatenate multiple messages to allow for longer content, but it is recommended to switch to IP-based transport when transmitting large data. Once signaled via SMS, devices could also power up additional subsystems and continue using IP connectivity and the normal UDP binding.

Another proposal is to run CoAP over TCP [115], which is interesting for backend clusters. Here, a few endpoints exchange CoAP messages over long-lasting TCP connections, for instance, when balancing the load among replicated systems. The main reason for this binding, however, is dealing with IPv4, and network address translation (NAT), firewalls. The timeout for keeping UDP ports open is often as low as 60 seconds. When using NAT, the source port is also likely to change whenever a mapping timed out. This means that the messages will look like they come from a new CoAP endpoint, which breaks observe relationships and requires session resumption for DTLS. For TCP, the timeouts are usually ten to twenty times longer. Firewall rules are also more relaxed, since they can make use of the TCP reset message to close the port for unwanted traffic. With UDP, it is harder to distinguish between actual traffic and possible flood attacks, since it would require deep packet inspection. Considering the expected amount of IoT traffic, it would be sensible to

[6] This example is only for illustration, as the exact URI scheme was not finalized at the time of writing.

make firewalls aware of DTLS and CoAP control messages (e.g., RST). Depending on its success, CoAP could be granted an exceptional position similar to that of HTTP in current firewall systems.

2.4 Service Description and Discovery

The CoRE working group also defined entities for service discovery. Using REST, all services are provided through Web resources that are accessed through an initial URI or bookmark. Thus, there must be a machine-readable description of resources, a mechanism to discover them on a server, and ways to discover individual devices.

2.4.1 CoRE Link Format

Web resources for IoT applications are described in the CoRE Link Format [168]. It is an extension of Web Linking [140], which was introduced to provide metadata for a resource in the HTTP header. An example is given in Figure 2.14; the line breaks are only for clarity and not part of the Link Format encoding. `title` is already defined in the underlying Web Linking specification and contains a human-readable name. The `rt` can provide a short tag to identify the semantics of a resource. These tags are shortcuts and must be known beforehand. Otherwise clients must be able to follow link that can be given in the `if` attribute. Usually, this attribute also carries a short, well-known tag [174]. `sz` can give an estimate of the maximum representation size and `ct` provides a list of the available representation formats (see Section 2.2.5). Machines can also figure out related resources, e.g., a shortcut at an alternative, usually shorter URI path. All attributes allow for filtering using queries, e.g., `/.well-known/core?rt=ucum:cel` for all resources that provide a temperature in degrees Celsius. However, only byte-by-byte comparisons including an asterisk as wildcard are defined. That means arithmetic comparisons (e.g., on the `size` attribute) are not possible.

```
</sensors/temp>;rt="ucum:cel",
</large>;sz=1280;title="Large resource",
</multi-format>;ct="0 41";title="text/plain and application/xml",
</t>;anchor="/sensors/temp";rel="alternate";title="Shortcut to sensor",
</semantic>;rt="restdesc";if="http://api.example.com/usage/"
```

Figure 2.14: The CoRE Link Format consists of links in angle brackets and attributes that are attached with a semicolon. The linebreaks after the comma separating the links were only added for readability and are not part of the format.

2.4.2 Resource Discovery

In the original approach for HTTP, the Web links are carried as metadata in each response through the `Link` header field [140]. Because this requires crawling and causes a considerable overhead in the messages of IoT devices, CoAP servers provide the CoRE Link Format separately as resource representation in its own Internet Media Type (`application/link-format`). Humans discover Web resources by browsing. They follow links starting from the root resource of a Web site.

For machines using CoAP, the initial URI on servers is `/.well-known/core`, a well-known URI following RFC 5785 [141]. The Link Format provided here can list all resources provided by the server as relative Web links (e.g., `/sensors/temp`), but can also link to other service entry point URIs that provide their own Link Format. In other words, the discovery mechanism of CoAP is hypermedia-driven, fulfilling the HATEOAS constraint of REST.

2.4.3 Device Discovery

The device discovery mechanism uses both the CoRE Link Format and the resource discovery mechanism. There exist two possibilities:

The first is to inquire the well-known discovery resource through the 'all CoAP nodes' multicast address, which will yield multiple responses with the CoRE Link Format. This is limited to link-local and site-local networks, but still might cause congestion through numerous responses. Thus, multicast discovery should only be used with filtering and response suppression by non-matching devices.

The second way is more scalable and reliable, but requires an additional entity: the CoRE resource directory (RD), which usually resides in the backbone network [170]. On start-up, devices register their resources (or selected service entry points) by POSTing their Link Format description to the RD. In addition, they include metadata about the whole CoAP endpoint in the URI query variables, such as a specific endpoint name or the lifetime of the registration. The latter can be seen as heartbeat interval with which the device needs to report in: if there is no re-registration within this time, the device will be removed from the RD. Dynamic resources can also be patched in later with an update of the registration. Using requests for the registration of resources consequently means that CoAP nodes implement both client and server role. This is also common for IoT applications [174] and feasible due to the shared message encoding of requests and response.

Client endpoints can look up URIs for services at the RD using similar queries as for `/.well-known/core` of a single node, but without causing traffic in the constrained networks. In return, they receive lists of absolute Web links (e.g., `coaps://sensor1.example.com:5684/sensors/temp`). In general, RDs provide a collection of Web links that include M2M-specific attributes or even simple semantic annotations. These can also be shared or crawled to realize larger directories similar to search engines of the traditional Web.

2.5 Integration with Existing Infrastructures

Despite being a new protocol, CoAP integrates seamlessly into the existing Web infrastructure. A key property for this is the transparent mapping to HTTP: When existing HTTP-based applications require IoT support, they can simply use a cross-proxy that converts between the two Web protocols. The RESTful methods, response codes, and media types have a one-to-one mapping. Thus, unlike application-level gateways that do protocol translation, these proxies are application-agnostic. They can remain untouched when the application is upgraded or new device types are added. Being in-line with the end-to-end arguments of the Internet is the key difference to previous gateways that have been connecting IoT technology to the Internet.

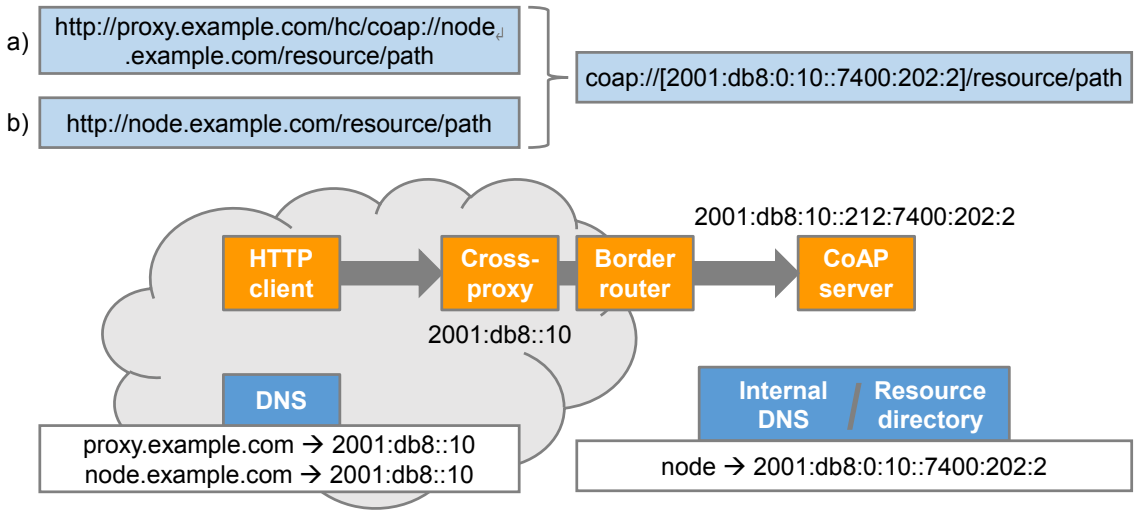


Figure 2.15: Both reverse and interception cross-proxy look like the origin server to the client. (a) depicts embedding the CoAP URI in the path for the reverse proxy. This allows for an automatic mapping when using a standard format. (b) show a URI that uses an intercepting proxy and virtual hosts that can be resolved through an internal DNS or RD.

Often the proxy is placed close to the CoAP servers and the clients are kept unaware of talking to a proxy. Caching in these intermediaries also shields the constrained network from too many requests from unconstrained Web clients. This makes the architecture scalable. There are two possible implementations for this scenario: reverse proxies and interception proxies.

A reverse proxy uses the layered system constraint of REST. It acts as a server, but forwards the requests to the origin server that actually holds the information. The proxy can also answer from its cache when a similar request was made earlier and the response is still fresh (i.e., `Max-Age` has not expired). Clients will discover URIs that logically point to an IoT device, but actually resolves to the reverse proxy. They can also use HTTP directly. In this case, the proxy requires a mapping from the HTTP URI to the URI pointing to the CoAP origin server. One approach is embedding the CoAP URI in the path of the HTTP URI [36], for instance:

`http://proxy.example.com/hc/coap://node.example.com/resource/path`

This can be combined with URI Templates [77], which also enables a generic translation service that can be used in Web mashups. The other one is using virtual hosts, that is, multiple DNS entries that all point to the reverse proxy. The latter then needs a mapping from its virtual hosts to the origin server. This can be provided by an internal DNS server or an RD using the endpoint identifier as hostname within the domain [170]. Both approaches are depicted in Figure 2.15.

An interception proxy is fully transparent to the clients. They send their requests directly to the origin servers, but this proxy intercepts them somewhere along the routing path. Instead of forwarding the IP packet of the original request, it issues a new, translated request to the origin server or directly answers from the cache, spoofing the address of the origin server. This requires the proxy to be on a node through which all traffic to the constrained network is routed or traffic redirection. Interception proxies have the advantage of zero configuration within clients and servers. They become problematic, however, when security mechanisms such as end-to-end authentication and encryption come into play. Devices have to export their private keys to the proxy, which might be stored in trusted platform modules (TPMs) by the manufacturer.

For the other direction, that is, CoAP nodes contacting HTTP services, it is often easier to use a forward proxy. Here, the clients are aware of going through a proxy and they encode this in the request. In HTTP/1.1 this is done by using an absolute URI instead of only the path in the Request-Line:

```
GET http://origin.example.com/resource/path HTTP/1.1
```

The authority of the URI thereby resolves to a different address than the TCP connection used for the request sent to the proxy. CoAP provides two options for requests issued to forward proxies:

- The `Proxy-Scheme` option works by forming an absolute URI together with the other Uri-* options. Currently, this is defined for the `coap`, `coaps`, `http`, and `https` schemes.
- The other option is `Proxy-Uri`, which holds a UTF-8 string of an absolute URI, where reserved and non-ASCII characters must be percent-encoded. This alternative has the advantage that links found in hypermedia need not to be parsed by a constrained node. It can simply copy the URI string—which can have any scheme—to this option and the proxy will take care of parsing and retrieving the representation. Therefore, the application-agnostic proxies also help to take other burdens from the constrained devices.

An open issue is the mapping of unique CoAP features, which will require common practices to emerge. There are several workarounds for push notifications in HTTP such as long polling [121], chunked transfer streaming [69], or custom protocols over Web sockets [68]. In the traditional Web, however, it is the application that defines which one to use, so cross-proxies cannot remain application-agnostic for this feature. The same applies for group communication, as HTTP does not support IP multicast.

This chapter summarized the basics of the new Web protocol suite that was designed in the CoRE working group, in which we participated. The remainder of this dissertation will now highlight our contributions for the different components that build up the IoT, namely resource-constrained devices (Chapter 3), cloud-based services (Chapter 4), and the human in the loop (Chapter 5).

Chapter 3

Resource-constrained Devices and Efficiency

Wireless sensor networks (WSNs) are a fundamental building block of the Internet of Things (IoT). With their sensors and actuators, they provide the interface to the physical world. In contrast to previous sensing systems, WSNs consist of tiny autonomous sensor nodes that are equipped with a processor, a low-power radio, and a power source. Thanks to the advances in microelectronics, microelectromechanical systems (MEMS), and wireless communications, these devices are inexpensive to produce and are able to run on batteries or energy harvesting. Through collaboration, they can provide data at much higher spatial resolution or efficiently monitor large areas. Initially, this made WSNs interesting for military applications such as the DARPA-funded *Smart Dust* project [99]. Soon after, civilian applications such as habitat and environmental monitoring emerged [18, 128, 195]. Due to the low costs and standards emerging at that time (e.g., IEEE 802.15.4 for low-power communication [1]), the technology was also adopted in commercial and industrial automation systems such as ZigBee or WirelessHART [12, 176]. This trend ultimately led to the IoT as perceived today, where smart objects actively communicate their state or await actuation commands.

Furthermore, the fundamental technology for the IP-based IoT originated in the research area of WSNs. While the initial innovation has largely been driven by advances in hardware design, software solutions now play the primary role for IoT applications by connecting devices to digital services. The resource-constrained sensor and actuator nodes, usually a Class 1 device (see Section 2.1), need protocols that are power-efficient on the one hand, but on the other hand allow for easy and flexible integration into existing IT systems. Here, 6LoWPAN with the narrow waist of IP plays a key role. For interoperability at the application layer, we provide one of the first implementations of the Constrained Ap-

plication Protocol (CoAP) that uses lightweight implementation techniques: Erbium (Er) for the Contiki operating system [48]. Its design progressively contributed to the protocol drafting process within the IETF CoRE working group, for instance, to optimize blockwise transfers for resource-constrained devices, improve the observe mechanism, and provide numbers to estimate the overall implementation complexity of CoAP. Conceptually, this chapter presents the *thin server architecture* and guidelines to scale Web technology down for direct use on resource-constrained devices. We recommend to adhere to the layered architecture of the Internet protocol stack to manage complexity and foster reliable and reusable protocol solutions. Furthermore, Erbium was the first CoAP solution to leverage a radio duty cycling (RDC) protocol to provide power efficiency. We experimentally evaluate our low-power CoAP, demonstrating that an existing application layer protocol can be made power-efficient through a generic RDC mechanism. Our results question the need for specialized low-power mechanisms at the application layer.

The contributions of this chapter are based on our publications [104], [106], [107], and [110]. We first introduce the related work on low-power IP including mechanisms for power-efficient operation as well as other CoAP solutions. Section 3.2 then presents our thin server architecture for the design of software for resource-constrained IoT devices. In Section 3.3, we give an overview over our Erbium implementation and discuss the lessons learned about CoAP. Our evaluation and results are discussed in Section 3.4.

3.1 Related Work

Our work aims at the seamless integration of resource-constrained devices into the World Wide Web. A prerequisite for this is having end-to-end IP connectivity. We first discuss the related work that provides the overall framework for our application layer design. Next, we give a brief introduction into the techniques to make embedded IP stacks energy-efficient, before we present comparable and alternative approaches to our thin server architecture.

3.1.1 The IP-based Internet of Things

The idea to connect resource-constrained devices directly to the Internet using the Internet Protocol (IP) originated in the field of wireless sensor networks (WSNs). Adam Dunkels and Zach Shelby independently showed that RFC-compliant IP support is feasible for resource-constrained sensor nodes [44, 173].[1] In 2006, the idea was picked up within

[1] The Contiki operating system by Adam Dunkels actually started as a retrocomputing project to run a Web server on a Commodore 64 computer. Only later, it was adopted for the use in WSNs for its excellent

Layer	Protocol
Application	CoAP / REST Engine
Transport	UDP
Network	IPv6 / RPL
Adaptation	6LoWPAN
MAC	CSMA / link-layer bursts
Radio Duty Cycling	ContikiMAC
Physical	IEEE 802.15.4

Figure 3.1: Low-power operation is implemented only in the RDC layer, thereby separating the concern from the application layer. This reduces complexity and follows the layered architecture that allowed the Internet to evolve.

the IETF, where a working group was chartered for *IPv6 over Low power Wireless Personal Area Networks (6LoWPAN)*. It focused on the IEEE 802.15.4 low-power radio specification [1], which has also been popular in the WSN community. While Contiki and NanoIP originally focused on IPv4, IPv6 turned out to be a better fit for networked embedded devices. The IP successor has a modular design that is good for extensions and its addressing scheme fits well with the 64-bit Extended Unique Identifier (EUI-64) for MAC-layer addressing of the new wireless technologies. Moreover, the huge 128-bit address space of IPv6 allows the assignment of global IP addresses to every single IoT device. This eases access and solves the NAT traversal problem [78], which often interferes with IoT applications.

The 6LoWPAN specifications [92, 135] define a small adaption layer between the MAC and the network layer (see Figure 3.1) that provides compression mechanisms for the large 40-byte IPv6 headers. They can become as small as 6 bytes, depending on the addresses used. In addition, 6LoWPAN defines a fragmentation mechanism, since the smallest MTU of IPv6 is 1280 bytes. The frames of IEEE 802.15.4, however, only carry 127 bytes. RFC 6775 [171] is also a result of the 6LoWPAN working group and defines an optimization for the neighbor discovery mechanism of IPv6. Recent efforts within the IETF focus on the expansion of the IPv6 to other low-power technologies such as DECT ULE and BLE [29, 58, 59, 139].

In academia, IP for WSNs took off with the implementations for *TinyOS* [90] and *Contiki* [55]. Many new applications such as *ACME* [96] or *sMAP* [41] emerged that used seamless IP connectivity, but still custom UDP-based application-layer protocols. Researchers also adopted other successful mechanisms from the field of WSNs and helped to standardize them for low-power IP. The *Collection Tree Protocol* [74] was the basis

handling of resource constraints.

for the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [183, 198] and *Trickle* [120] is used in the Multicast Protocol for Low power and Lossy Networks (MPL) [91]. More recent work focuses on the improvement of the reliability on low-power IP networks. *ORPL* improves the original RPL with opportunistic forwarding and achieves higher goodput and lower latencies [53]. The *Low-power Wireless Bus* uses constructive interference and the capture effect to enable bus-like flooding of wireless networks [67]. This approach was not designed with IP communication in mind, however, because it is highly sensitive to timings and requires homogeneous hardware platforms. Yet there are efforts to enable TCP for a seamless connection with the outside world [86]. Due to the high reliability and low latency of the Low-power Wireless Bus, TCP is expected to perform well in this environment. Finally, the *OpenWSN* project aims at the integration of a standards-based mechanism for energy-efficiency based on the time slotted channel hopping (TSCH) mode of IEEE 802.15.4e [187, 190]. At the time of writing, this approach is being standardized in the IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) working group in the IETF. The nodes will come with an integrated CoAP implementation, since it is used to communicate the configuration parameters among nodes.

3.1.2 Power-efficient Protocols

On typical IoT platforms, the radio transceiver is one of the most power-consuming components. Idle listening is as expensive as receiving packets. To conserve energy, the radio transceiver must be switched completely off for most of the time.

Sleepy Nodes

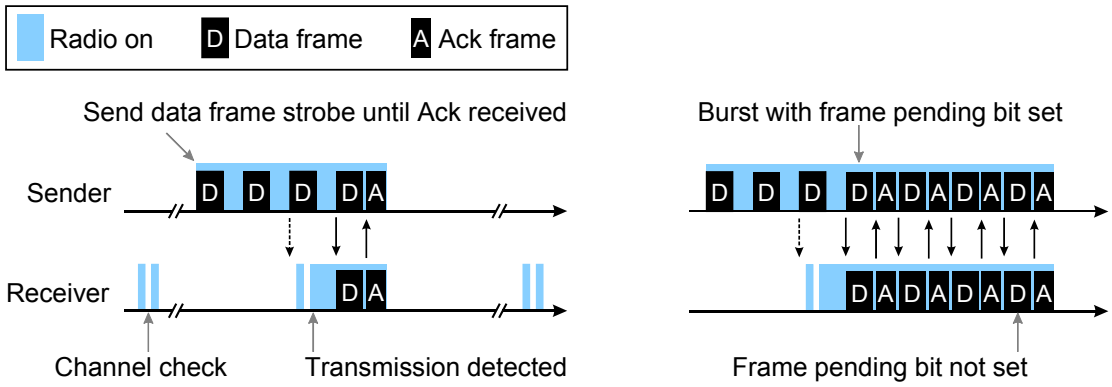
Industry has been focusing on an extended sleep mode mechanism, often referred to as 'sleepy nodes.' The sleep periods can range between seconds and weeks and is fully controlled by the application layer. The business logic (i.e., the code that implements the actual functionality on top of the operating system (OS) and protocols) can define suitable moments to go to sleep mode (e.g., after data was uploaded to the cloud) and on which events to wake up (e.g., a sensor stimulus). This strategy is particularly useful for communications other than IEEE 802.15.4 such as Wi-Fi or cellular. Low-power Wi-Fi, for instance, is mainly based on long sleeping periods and short wake-up cycles [143]. Often devices have to re-synchronize and maybe re-associate with the network, though. For this, IEEE 802.11v [4] provides power save services to coordinate the sleeping intervals. Since the devices are not available during sleep mode, the business logic is burdened, too, to maintain connectivity and continuous operation. Furthermore, it is hard realize a response infrastructure that can be inquired and re-configured at any time.

Radio Duty Cycling

In contrast, RDC strategies provide virtual always-on semantics that are transparent for the application [32, 49, 56, 153, 203]. Several RDC algorithms have been designed, allowing nodes to keep the radio chip off for more than 99% of the time while still being able to send and receive messages [49, 56].

In this work, we use the *ContikiMAC* RDC protocol [45]. ContikiMAC is a low-power listening MAC protocol that uses an efficient wake-up mechanism to attain a high power efficiency: with a wake-up frequency of 8 Hz, the idle radio duty cycle is only 0.6%. Its principle of operation is shown in Figure 3.2. Nodes periodically wake up to check the radio channel for a transmission from a neighbor. If a radio signal is sensed, the node keeps the radio on to listen for the packet. If the frame is not addressed to the checking node, it goes back to sleep. When the data frame is correctly received, though, the receiver sends an acknowledgment frame. To send a packet, the sender repeatedly sends the data frame in a so-called strobe until it receives an acknowledgment, or until the packet was sent for an entire channel-check interval without an acknowledgment being received. This can be optimized by storing the wake-up times for each neighbor. The so-called phase lock starts the strobe just before the receiver awakes and minimizes the active transmit time, which also reduces channel utilization.

ContikiMAC can easily be combined with *link-layer bursts*. When a sender has several frames to send, it first wakes up its neighbors with a ContikiMAC strobe and sets the *frame pending* bit in the 802.15.4 frame header to tell the receiver that another frame will follow.



(a) ContikiMAC strobe (b) Link-layer bursts

Figure 3.2: (a) ContikiMAC senders wake their neighbors up by sending a strobe of data frames until receiving an acknowledgment. Only the addressed node stays awake to receive the frame. (b) Link-layer bursts use the frame pending bit in the 802.15.4 frame structure to signal the receiver to keep the radio on.

3.1.3 CoAP Solutions

By now, there are many implementations of CoAP for IoT devices, such as *libcoap* [111], the *NanoService C Device Library*[2], *SMCP*[3], *microcoap*[4], *CoAPSharp*[5], *TinyCoAP* [123], and a TinyOS-based CoAP with Efficient XML Interchange (EXI) support [35]. Yet none has been evaluated in terms of low-power behavior in a multi-hop network. Kuladinithi et al. [111] measure the latency of their CoAP implementation, but without any duty cycling and only in a single-hop network. Colitti et al. [40] take a first step toward analyzing the power consumption of an earlier CoAP implementation for Contiki [202], but with a simplified power model. It only considers application data size and that does not include the full energy consumption by the system. By contrast, we are the first to experimentally evaluate the full system power consumption of a multi-hop low-power IPv6 network [107]. The *OpenWSN* project evaluated TSCH in a 2-hop network using CoAP only recently [187].

3.1.4 Alternative IoT Protocols

CoAP is not the only protocol that aimed at a Web-like application layer for resource-constrained devices. Embedded Binary HTTP (EBHTTP) [180] was an attempt to compress the verbose Hypertext Transfer Protocol (HTTP) and send a more compact format over UDP. It was used in the initial sMAP project [41]. *RESTful Contiki* [202] implemented an HTTP API for IoT applications. Its performance, however, suffers from the problems discussed in Section 1.1. The *Smews* Web server [51, 52] is able to overcome these problems through cross-layer optimization. Using pre-compiled TCP segments and efficient connection management, it is able maintain up to 256 concurrent TCP connections with a respectable performance at the transport layer. Yet all constrained HTTP implementations only mimic the protocol to be interoperate with standard HTTP libraries that are used in backend systems. None of them provides the rich feature set of the REST architectural style such as content negotiation and cache control.

An alternative to Web-like protocols is the Message Queuing Telemetry Transport (MQTT). Instead of REST, it follows the publish–subscribe messaging pattern. The protocol originated at IBM and was initially used for pipeline monitoring. It recently became an open standard within the OASIS consortium [11]. MQTT is also based on TCP, though, while using a compact binary format. The main drawback of this publish–

[2] http://mbed.org/ (accessed on 12 Feb 2015)
[3] https://github.com/darconeous/smcp (accessed on 12 Feb 2015)
[4] https://github.com/1248/microcoap (accessed on 12 Feb 2015)
[5] http://www.coapsharp.com/ (accessed on 12 Feb 2015)

subscribe protocol is missing extensibility. Like HTTP clients for the IoT, MQTT clients must also be pre-configured with a dedicated service. This tight coupling also makes it hard to adapt to an evolving environment.

3.2 The Thin Server Architecture

WSNs have been a precursor to the IoT, but with different system requirements for applications. They usually consist of one homogeneous network with a dedicated task. In the IoT, different application domains converge, resulting in networks of heterogeneous devices and changing applications. Hence, there is the need for a software architecture that takes the dynamics of IoT applications and the requirements of resource-constrained devices into account. To this end, we propose the *thin server architecture*, which enables application-layer interoperability at the device level and IoT service composition through loosely coupled mashups.

3.2.1 Design Goals

Amazon, Google, and Facebook as well as the other big players in the Web sphere define their public APIs independently. They are still able to interconnect seamlessly to provide services of higher value, for instance, a hotel booking site that integrates a map service. Also for third parties, the integration of different services is comparatively easy. This is possible because of the REST constraint for uniform interfaces. We aim at having the same properties for resource-constrained devices in the IoT.

Furthermore, we want to enable multiple applications to co-exist at the same time in the same environment. Traditionally, all nodes in a sensor network run the same software because they all serve the same application. In the IoT, devices are heterogeneous and different applications overlap in the same network, while the ultimate goal is convergence for both networks and applications. Our goal is to have an IoT device infrastructure that is flexible, can respond to operational and diagnostic inquiries at any time, and can serve multiple running applications in parallel.

Finally, we want to lower the entry barrier for IoT application development. Programming wireless sensor nodes is hard because the constrained resources prohibit the usual abstractions applied to distributed systems. There are solutions from the field of WSNs for domain-specific abstractions, but they lack the generality required for the vision of the IoT. Our architecture supports a Web-like programming model that enables developers from different backgrounds to create and customize IoT applications.

3.2.2 Separation of Application Logic

Usually, the hardware of small IoT devices does not change once they are deployed. Finished products, such as consumer electronics or calibrated industrial sensors, do not have extension slots like the rapid prototyping platforms prevalent today. It is only the application logic that needs to adjust to new requirements. Thus, we propose a separation of application logic and elementary device functions. The device firmware then can remain immutable (i.e., 'firm') for every kind of application as long as all hardware functionality is accessible through an all-purpose interface. An implication of this architecture is that the device utilization can continuously evolve during its lifetime, as new functionality can be added by providing a new 'app' that is independent from the hardware and computing power of the device. This also enhances the environmental sustainability, as the use of hardware can be progressively extended.

The application logic can run on any machine connected to the network, which usually means the Internet. One aspect of cloud computing is the Software as a Service (SaaS) model, where it becomes irrelevant for users where the application code is running. This allows for efficient hosting of services on compute farms near power plants, since transferring information is cheaper than transferring energy. In the IoT, however, some applications have critical timing requirements, for instance, control loops for automation tasks. This requires a careful placement of application logic within the network. Fog computing is a newly defined research field that focuses on cloud-like techniques for services that need to be hosted close to the operational network to meet real-time requirements—hence the term fog, which is closer to the ground than clouds [21, 185]. In this thesis, the placement of application logic is out of scope and we use the term cloud in its general sense that the service may be hosted anywhere, given the application requirements are met.

3.2.3 Thin Servers

IoT devices possess the data corresponding to the state of the physical world. Hence, they are best modeled as RESTful origin servers, for which we propose the following properties. Based on the idea of *thin clients*, we define a *thin server* as a device in the role of a server that does not host any application logic. Corresponding to a thin client, which is only equipped with the necessary interfaces to interact with the user such as a display and a keyboard, thin servers only provide the general-purpose API to their elementary functions:

- Sensor and actuator access to interact with the physical world
- Device management

These RESTful hardware wrappers require developers to implement Web resources that control general-purpose input/outputs (GPIOs) or analog-to-digital converters (ADCs) and serialize the exchanged representations. The critical parts of the embedded software (e.g., the core device drivers and network stack) are reused. This lowers the demand for low-level system experts—who are usually scarce—to maintain the firmware.

Originally, the Web of Things has been using HTTP clients to implement IoT devices. This is due to the lack of a push notification mechanism. Low-power IoT devices usually do not maintain a continuous connection and only send data when an event occurs. This can easily be modeled with clients that send POST requests. These need to be pre-programmed with a destination URI, though, and lack the ability of flexible bindings. While our architecture is also applicable to HTTP, we assume thin servers to be implemented with CoAP, which provides push technology as well as group communication.

3.2.4 Application-agnostic Infrastructure

The separation of application logic and device firmware has further benefits. Traditionally, sensor and actuator nodes are programmed exclusively for a single application that comes with its own abstractions. One the one hand, this allows for highly optimized network behavior such as in-network processing. On the other hand, the device infrastructure can only serve a single purpose, as it is simply not feasible to install multiple applications on a resource-constrained device. The common solution to connect different applications, is to use an application-level gateway for protocol translation. This, however, creates the vertical silos that cause high integration costs and prohibit interoperability of heterogeneous devices.

When all devices only provide their elementary functions, they all have the same, low abstraction level. The connected IoT devices then become an application-agnostic infrastructure that can be used by any application and at the same time. Thereby, it does not matter if the client of a RESTful device resource is a local node in the low-power network, an application running on the border-router, or a remote cloud service. The well-known patterns from the Web can be applied in every part of the system, which allows for better understanding, maintainability, and usability of applications. Overall, this enables the grand platform for the IoT, as there is no vertical integration. The programming model becomes similar to Web mashups, which allow a lightweight composition of different services to realize applications [138].

3.2.5 Open Marketplace

In turn, the device infrastructure opens up the support of open marketplaces. Third parties can provide software for devices that improves their functionality or interplay. Application modules—or 'apps' with reference to the successful iOS App Store or Android Play Store—can be reused for many deployments, for instance for data collection or automation of a specific device. As observed in the Web or the smartphone market, independent developers yield a lot of creativity. Hence, open marketplaces can help driving the innovation in the IoT. The separation from the firmware also allows developers that are not specialized in the embedded domain to create applications. They can fully focus on the functional requirements, while resorting to the programming environment of their choice.

This also requires a common runtime environment for the application modules. In the World wide Web, mashups mainly run in the Web browser, which provides a common API. With the success of server-side JavaScript, a similar environment can be provided for faceless IoT mashups. More details on this topic are provided in Section 5.2.

3.2.6 Intuitive APIs

To ease the job for developers, our design promotes human-readable, self-descriptive REST APIs that are supported by concise, machine-readable descriptions such as the CoRE Link Format [168]. Web resources such as `coap://node1.example.com/sensors/temp` with meaningful representations such as SENML [95] do not require further documentation to be understood, while well-defined Internet Media Types and link relations tell machines how to interpret the data [71].

This is a major difference to alternative architectures such as *Open Mobile Alliance (OMA) Lightweight M2M* [122], which is also used by the *IPSO Alliance*[6]. LWM2M uses a resource-oriented architecture, but with pre-defined URI paths that encode specific function sets. Further, the path segments are string encoded decimals (e.g., `/3/0/13`), which reduces the powerful URIs concept to numeric service identifiers. Developers and operators are doomed to refer to out-of-band documentation to understand the system and errors are harder to find. Hard-coded URIs also violate the HATEOAS concept of REST.

Our approach does not only support professional developers, but also enables tech-savvy users to understand how their devices interact. Although most functionality of today's consumer devices is realized in software, users are usually limited to use products as designed by the manufacturers. By combining an intuitive interface to the elementary

[6] `http://www.ipso-alliance.org/` (accessed on 12 Feb 2015)

functions of a device with scripting support for the applications, users can change and extend its functionality and smoothly integrate it with the existing infrastructure without modifying the original device firmware. This adds significant value to the product since users can adopt their devices to their specific needs. Moreover, according to Metcalfe's law [133], publishing the added functionality increases the value of the whole infrastructure of connected devices.

3.3 Erbium (Er) Implementation

We have implemented our thin server architecture with a low-power CoAP framework for the Contiki operating system. Our *Erbium (Er) REST Engine* is extending the *Contiki REST Layer* by Yazar and Dunkels, which provides a generic abstraction for RESTful applications [202]. Our implementation is written in C and is optimized for microcontroller platforms with constrained resources. It is available from the official Contiki Git repository[7]. While we apply several techniques to lower the memory footprint, we also focus on a developer-friendly API. Developers implement application logic similar to classic Web frameworks and only need to provide functions for the Web resource handlers. Being one of the first publicly available C implementations of CoAP, it was also ported to Linux for more powerful systems and larger frameworks.[8][9][10]

3.3.1 REST Engine Overview

Our CoAP implementation is build around a REST Engine that provides a Web-like abstraction for the user. It is inspired by the *Contiki REST Layer* of the Contiki Projects community[11]. We provide an improved Web resource abstraction that reflects common resource types. Furthermore, we implemented interfaces, so that the REST Engine can be either backed by our Erbium CoAP implementation or an HTTP implementation as depicted in Figure 3.3. The network-stack-like model provides the necessary mappings of RESTful methods, status codes, header options, query variables, and so forth to their CoAP and HTTP representations. This way, the application code is decoupled from the underlying protocol.

[7] https://github.com/contiki-os/contiki (accessed on 12 Feb 2015)
[8] http://www.coap.or.kr/ (accessed on 12 Feb 2015)
[9] https://www.eclipse.org/wakaama (accessed on 12 Feb 2015)
[10] https://github.com/mcollina/node-coap (accessed on 12 Feb 2015)
[11] http://sourceforge.net/projects/contikiprojects/ (accessed on 12 Feb 2015)

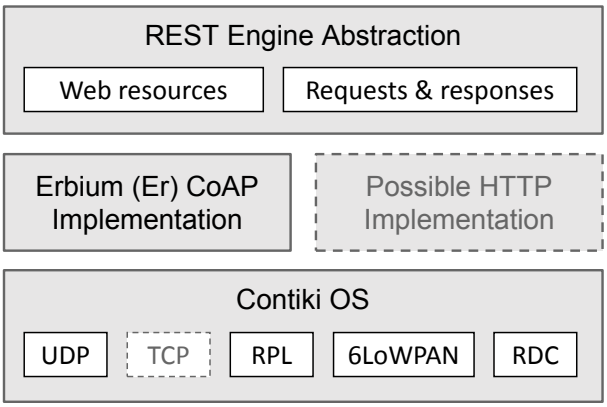


Figure 3.3: Erbium provides the CoAP implementation within the REST Engine abstraction. Following Contiki's network stack model with a struct holding pointers to functions and constants, the REST Engine can also be linked to an HTTP implementation. The latter is out of scope of this work.

The CoAP specification is modular, which allows for optimizing the memory footprint by only including the features required for the elementary device functions. Thus, we structured our implementation accordingly and every feature has its own C module:

- `er-coap` implements the core of the protocol such as message parsing and serialization. The protocol definitions are separated into two header files: `er-coap-constants` for the fixed definitions and `er-coap-conf` for the parameters that can be tweaked for special application requirements.
- `er-coap-engine` provides the control flow for client and server behavior. Since Erbium implements the thin server architecture for resource-constrained devices, the engine primarily takes care of calling the user defined resource handlers and automatic exception handling such as bad requests or internal server errors.
- `er-coap-transactions` enables reliable transmissions and allocates the required timers and message buffers.
- `er-coap-separate` implements convenience functions to store and retrieve state storage for the split-phase execution of separate responses.
- `er-coap-observe` takes care of managing the list of observers as well as creating and sending the notifications.
- `er-coap-block` provides helper functions for request and response fragmentation.
- `er-coap-res-well-known-core` is an implementation of the discovery resource. It automatically creates the CoRE Link Format for all activated Web resources. The filtering feature can be disabled to save memory.

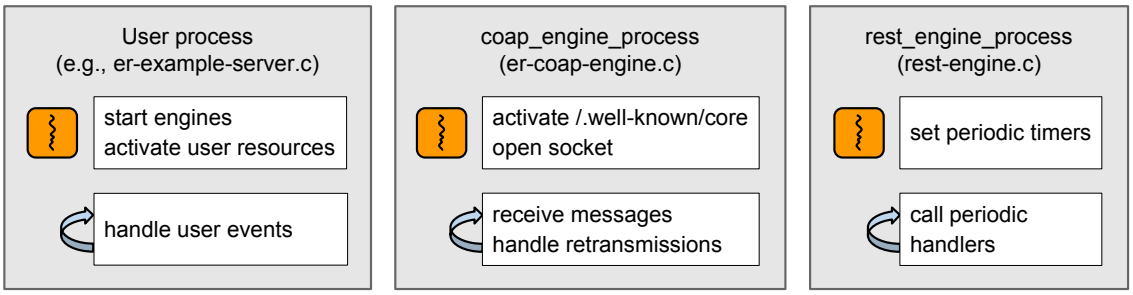


Figure 3.4: Erbium runs in three processes, which use collaborative multi-threading in Contiki. Each process has initialization tasks and a event handler loop.

Erbium uses three Contiki processes as shown in Figure 3.4. Like every Contiki project, there is a user process that defines the application. To use Erbium, the application must start the REST Engine, which will automatically start the protocol engine used. After that, the user process is used to activate the desired Web resources and to handle any user-defined events, which might trigger notifications. Each engine runs in its own (protothread-based [50]) process to allow for decoupled protocol implementations. The CoAP engine activates the discovery resource under `/.well-known/core` and opens a UDP socket on the configured port. After that, it will loop to react to network events (i.e., incoming messages) and to elapsed retransmission timers. The REST Engine takes care of periodic timers that are used by specific Web resource types.

Web Resource Types

We found five Web resource types that reflect the common patterns in RESTful IoT applications.[12] Erbium provides the following preprocessor macros to create user-defined Web resources based on these types:

RESOURCE A basic Web resource is defined by a name, a CoRE Link Format description [168], and the functions that implement the resource handlers for GET, POST, PUT, and DELETE. If a null pointer is passed for a function, the protocol engine will automatically return a `Method Not Allowed` response code. The handler functions receive the request and generate the respective response. To abstract from the protocol implementation, the REST Engine provides convenience function to access the messages (e.g., `REST.set_header_etag(response, etag_buf, etag_len)`). An example of a basic resource implementation is shown in Listing 1.

[12] Having gained more experience during projects, we refined our original design published in [107].

PARENT_RESOURCE Often a resource handler should intercept all requests to its child resources, for instance, when using URI Templates [77] or when the resource path maps to a database-like internal structure. This is provided through the PARENT_RESOURCE type, which is otherwise similar to the basic resource type. The handler function, however, needs to evaluate the URI path programmatically to create the corresponding response.

EVENT_RESOURCE This abstraction requires an additional handler function that reacts to custom events. This can be, for instance, a button press or a PUT request that caused a status update. The handler must be provided by the developer and connects the event to the data model of the resource. It can evaluate the change and, if desired, use the Erbium API to notify the observers of this resource (see Section 2.3.1). Adding and removing observers is handled automatically for this resource type.

PERIODIC_RESOURCE Most sensing tasks require the periodic sampling of the sensor. For this, the developer can define the sampling interval of a `PERIODIC_RESOURCE` with which the REST Engine will poll a handler function similar to the previous event handler. The periodic handler can call the sensor API and cache the reading for normal GET requests. Usually, it is also used to notify the observers based on custom rules such as a threshold for value changes. The observers are again managed automatically.

SEPARATE_RESOURCE The last abstraction helps with the implementation of separate responses (see Section 2.2.3). They are usually used when the server requires some time to answer a request (i.e., longer than $T_{proc} = 2\ s$). This can be due to slow sensors, such as magnetometers that consume too much energy for continuous operation and need to be initialized every time, long-lasting computations, or communication with sub-systems that are connected via universal asynchronous receiver/transmitter (UART). To this end, the `SEPARATE_RESOURCE` provides buffers to store the state for a split-phase execution. The developer needs to provide a handler function that is triggered by the long-lasting task upon completion to create and send the response.

Each Web resource is implemented in its own C module and can be reused in multiple projects. The resource path is only defined when activating the resource. Through additional activation calls, the same resource implementation can be made available under multiple paths, for instance, to provide alternate shortcuts. All resources that share the implementation will refer to the same instance in memory. In case multiple, independent instances are required, the C module needs to be duplicated with individual resource names or the implementation needs to take care of multiplexing the variables of the resource.

Blockwise Transfers

Erbium also supports blockwise transfers, which are discussed in detail in Section 2.3.2. For an HTTP implementation, the discussed technique would refer to the maximum segment size of TCP, since the problem of limited buffers is the same. When implementing a server, the developer has to define the maximum payload size that can be buffered (`REST_MAX_CHUNK_SIZE`). This is usually determined by the target platform and is often much smaller than the 1280-byte MTU. Yet `REST_MAX_CHUNK_SIZE` is usually defined so that most resource handlers can fully generate their response within this buffer size—exceptions are the discovery resource and other bulk data transfers. In case a client requests a smaller block size, our engine fragments the response automatically. Hence, there is usually no need for the resource handler implementations to be made aware of this mechanism. To enable on-the-fly processing of larger data, the REST Engine passes the current block (or segment) number, byte offset, and preferred chunk size to the resource handler. The implementation can then use the helper function from `er-coap-block` to process incoming blockwise requests (`Block1` option) or to generate large blockwise responses (`Block2` option).

Observing Resources

CoAP's observe mechanism [84] is one of the most powerful features of the new Web protocol and for the thin server architecture. When the `Observe` option is set, the CoAP engine automatically registers observers for the two observable resource types. An `EVENT_RESOURCE` requires a handler function that can be called in the user process for any event. The handler function for the `PERIODIC_RESOURCE` is called automatically with the defined sampling interval. Both functions are used to implement the rules when the list of observes should be notified. A protocol-independent `REST.notify_observers()` function can then be used to push changes of the resource state to all observers of the resource. Our CoAP engine takes care of the transmissions.

CoAP Clients

Erbium also provides an API to implement CoAP clients. Sending requests is, for instance, required to register the provided Web resources with a CoRE resource directory [170]. We provide a blocking function (`BLOCKING_REQUEST()`) implemented with Contiki's protothreads [50] to issue a request and receive the response. This linear programming model can also hide blockwise transfers, as it continues only when all data were received.

3.3.2 API Examples

The following listings give an idea of the Erbium API. Listing 1 shows how a REST resource is implemented with our Web-like abstractions for resource-constrained devices.

The code for the user process depicted in Figure 3.4 is shown in Listing 2. The example is only skipping the includes of the header files. Otherwise, this short snippet results in a fully functional CoAP server with three resources: the *Hello World* resource from Listing 1, an `EVENT_RESOURCE` called `res_event`, and the discovery resource.

The REST Layer abstraction is meant for servers only. Hence, clients directly use the CoAP-specific API provided by Erbium. The minimal example in Listing 2 retrieves the CoRE Link Format from `coap://[aaaa::212:7402:2:202]:5683/.well-known/core` and prints the incoming data to the standard output stream. Usually, the response handler would implement the business logic or perform on-the-fly processing for blockwise transfers.

```
/* ... */
RESOURCE(res_helloworld,
  "title=\"Hello World resource\";ct=\"0 50\"", /* see CoRE Link Format */
  res_get_handler,
  NULL, /* or a res_post_handler */
  NULL, /* or a res_put_handler */
  NULL  /* or a res_delete_handler */
);

static void res_get_handler(void *request, void *response, uint8_t *buffer,
    uint16_t preferred_size, int32_t *offset)
{
  unsigned int accept = -1;
  REST.get_header_accept(request, &accept);

  if(accept == -1 || accept == REST.type.TEXT_PLAIN) {
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
    const char *msg = "Hello world";
    REST.set_response_payload(response, msg, strlen(msg));
  } else if(accept == REST.type.APPLICATION_JSON) {
    REST.set_header_content_type(response, REST.type.APPLICATION_JSON);
    const char *msg = "{\"message\": \"Hello world\"}";
    REST.set_response_payload(response, msg, strlen(msg));
  } else {
    REST.set_response_status(response, REST.status.NOT_ACCEPTABLE);
    const char *msg = "Supporting content-types text/plain and application/json";
    REST.set_response_payload(response, msg, strlen(msg));
  }
}
```

Listing 1: Web resources are agnostic of the protocol implementing REST.

```
/* ... */
extern resource_t res_helloworld, res_event;

PROCESS_THREAD(er_example_server, ev, data)
{
  PROCESS_BEGIN();
  rest_init_engine();
  rest_activate_resource(&res_helloworld, "/hello-world");
  rest_activate_resource(&res_event, "/observe");

  while(1) {
    PROCESS_WAIT_EVENT();
    if((ev == sensors_event) && (data == &button_sensor)) res_event.trigger();
  }
  PROCESS_END();
}
```

Listing 2: The main C module of a Contiki project defines the user process, which activates the individual resources and binds them to a path.

```
/* ... */
uip_ipaddr_t server_ip;
coap_packet_t request[1];

PROCESS_THREAD(er_example_client, ev, data)
{
  PROCESS_BEGIN();
  coap_init_engine(); /* not rest_ because CoAP-only */
  uip_ip6addr(server_ip, 0xaaaa, 0, 0, 0, 0x0212, 0x7402, 0x0002, 0x0202);
  coap_init_message(request, COAP_TYPE_CON, COAP_GET, 0 /* automatic MID */);
  coap_set_header_uri_path(request, "/.well-known/core");
  coap_set_header_accept(request, APPLICATION_LINK_FORMAT);
  COAP_BLOCKING_REQUEST(&server_ip, UIP_HTONS(COAP_DEFAULT_PORT), request,
      client_response_handler);
  /* error handling if client_response_handler() did not change */
  /* ... */
  PROCESS_END();
}

void client_response_handler(void *response)
{
  const uint8_t *chunk;
  int len = coap_get_payload(response, &chunk);
  printf("%.*s", len, (char *)chunk);
}
```

Listing 3: Client calls are implemented through the `COAP_BLOCKING_REQUEST()` macro. Using protothreads, it makes the asynchronous communication of CoAP look like a synchronous, blocking function call. When it returns, the client knows the request was executed. Timeouts and other errors can be handled through a global state variable that is modified by the response handler.

3.3.3 Lessons Learned

Implementing all major draft versions during the standardization of CoAP gave us valuable insights on the topics listed below. On the one hand, these learned lessons influenced the protocol design within the CoRE working group and, on the other hand, resulted in guidelines for efficient implementations of CoAP, which are also collected in an IETF document [106].

Message Processing

There are two strategies for processing CoAP messages: on-the-fly processing and internal data structures. The advantage of the former is that no additional memory is allocated by default. When used, numeric options need to be extracted to local variables due to their variable length encoding and potentially differing byte order. The savings in memory are paid with a higher processing effort. The encoded options need to be traversed once to check for unsupported critical options and then every time an option is checked and read. The latter can be optimized by maintaining a bit vector of the options present in a message. Due to the wide and sparse range of option numbers, the indices for the bit-vector cannot be based on left-shift operations. Hence, an implementation-specific enumeration of supported options should be used to mask the present options in the vector. In addition, a direct pointer to each option can be added to a sparse list for fast retrieval. The drawback of the on-the-fly strategy lies in message generation due to the delta encoding of option numbers: either the user needs to be restricted to set options in the correct order or the option list needs to be completely rewritten for each out-of-order update.

The second strategy is to allocate an internal data structure and parse all options to an internal representation while doing the initial check for unsupported critical options. This has the advantage that the numeric options are already in a variable of corresponding type for evaluation (e.g., a 16-bit integer in host byte order). The incoming payload and byte strings can be accessed directly in the encoded message using pointers. This approach can also benefit from a bit-vector of present options. Otherwise special values must be reserved to encode an unset option in the data structure. This might require more bits than required for the actual value range (e.g., a 32-bit integer instead of 16-bit). The second advantage is that users can set the options in arbitrary order, as the delta encoding is only applied before sending. This convenience is paid for with an increased memory footprint. Nonetheless, we chose this approach for Erbium to improve usability.

Memory Management

Contiki's cooperative multi-threading allows us to provide access to incoming payloads and byte strings (e.g., the ETag or query variables) directly in the IP packet buffer. Numeric header options are parsed and stored in additional integer variables of a message struct. This eases read and write access for the application, as CoAP uses a special differential encoding with variable lengths for options. For the response generation, Erbium originally organized a buffer for the resource handlers. This buffer was also reused to serialize the CoAP message in-place. The buffer size is defined by the `REST_MAX_CHUNK_SIZE` and `COAP_MAX_HEADER_SIZE` estimates discussed above. For existing wireless sensor node platforms, the maximum chunk size is usually 128 bytes or 256 bytes. Many applications even prefer 64 bytes to fit the CoAP messages into a single IEEE 802.15.4 frame. This of course, requires compact representations such as short plaintext values or the binary Concise Binary Object Representation (CBOR) [24] or EXI [100] encodings.

The additional buffer for response generation is only required for convenience and leaves room for optimization. In most cases, the server responds with an Acknowledgement (ACK) message, which is sent unreliably (i.e., without retransmissions). If we constrain the resource handler implementation to work in two phases, we can reuse the IP packet buffer for the response. The implementation needs to process the request first and store all relevant values in local variables. This is expected to cost far less memory than storing the whole request. Second, the response is generated by overwriting the request in the IP buffer. The IP and UDP headers remain untouched and can provide the source address. The in-place processing saves additional application layer buffer and significantly reduces the RAM requirements.

For separate responses and observe notifications, a server also needs to transmit Confirmable (CON) messages, which require a retransmission buffer. For embedded systems, it is preferred to refrain from dynamic memory allocation to ensure a deterministic behavior. Thus, we chose to allocate the corresponding buffer statically together with the Web resource that produces the response. This also helps modularity, since allocation is only required if the server makes use of CON messages. Normal resources can reuse the IP packet buffer and do not require any additional buffers. Resources with separate response allocate from a dedicated buffer pool. Note that resources that require such pool are prone to denial-of-services (DoS) attacks because resource-constrained devices can only afford a small number of message buffers. If simultaneous requests from different clients produce the same representation, the pool size can be set to one. However, the SEPARATE_RESOURCE still needs to store each client address and token in its state buffer during the split-phase execution. A similar strategy applies for observable resources.

Observing Resources

Usually, resource-constrained devices support only a few different Internet Media Types for the resource representations.[13] When observable resources only provide one content format, the notification process becomes very efficient: the server can send multiple CON notifications with a single retransmission buffer. The notification is serialized into this buffer, while destination address, port, token, and retransmission timers come from the list of observers. There, the endpoint address and token needs to be stored anyway for managing the list. This strategy requires a few `memmove` operations, but the the memory savings are significant so that a higher number of observers can be supported.

Blockwise Transfers

Originally, blockwise transfers were designed with a change of the initiative to the server for blockwise notifications.[14] That meant that a server needs to push all blocks of a notification to all observers. First, this causes a sudden burst of many packets in the LLN and the server must implement a congestion control mechanism.[15] Second, the server is burdened with the management of multiple ongoing transfers where each observer might need different blocks at a time. In case frequent changes of the resource state tend to interfere with the blockwise transfers (i.e., later blocks will belong to a new representation), the server can resort to atomic transfers [25]. This requires a large buffer for the complete notification body, though. In any case, it is highly recommended to use the `ETag` option for blockwise notifications.

The change of the initiative can be avoided when the server only sends the first block as notification and observers then use normal GET requests without the `Observe` option to retrieve the remaining blocks. The management overhead is now distributed among the clients and the server can use the optimization presented in the previous section. This change was enabled by pinning an observe relationship to the token. Previously, any GET requests without the `Observe` option canceled a relationship with the targeted resource. This was revised because it was prone to accidental cancellation, especially in multitenancy CoAP endpoints.

[13] This is also why we simplified CoAP's `Accept` option to be non-repeatable. It was the only repeatable uint option and required additional code for an mostly unused feature. Trial and error for the supported content formats can be avoided by listing them in the CoRE Link Format.

[14] Reversing the initiative was also used for combined `Block1-Block2` exchanges that are required for blockwise action responses to blockwise POST requests.

[15] Usually, congestion control is done at the client side by limiting the number of open requests.

Message Deduplication

Dededuplication provides only-once semantics for channels that might unintentionally create message duplicates. In CoAP, it is based on filtering and requires the receiving node to store the message identifier (MID) and the remote endpoint address, which is quite large due to the size of IPv6 addresses. For servers, each filter entry should also contain the original response, in particular when it is a temporary action result for a request that must not be executed again. This state must be kept for the lifetime of the respective request message, which is about 4 minutes (see Section 2.2.2). Since the amount of memory is scarce on resource-constrained devices, they can only maintain a limited number of entries. Thus, depending on the amount of available memory, deduplication can severely limit the number of interactions with a server. Because of tokens, clients usually have an explicit list of the responses they are expecting and remove the entry the first time they handle the corresponding message. All responses with unexpected tokens can be rejected.

Thus, the deduplication mechanism is mainly needed in CoAP servers to prevent multiple executions of the same request. One possible optimization that comes to mind is minimizing the time an entry needs to be stored. A server could always resort to separate responses to receive a confirmation. Keeping deduplication state together with its response is similar to storing the state for CON retransmission (remote endpoint address, message, and a timer, ignoring the retransmission counter). The latter, however, can be removed as soon as the acknowledgement arrives. Using an efficient three-way CON–CON–ACK request-response exchange that makes use of implicit acknowledgements (see Section 2.2.2) would inform the server when a client has stopped the CON transmission. Since messages can also be duplicated by the network, however, servers still need to keep a minimal filter entry with the MID and remote endpoint address, but without the response.

Our current implementation makes use of properties that allow to relax duplicate filtering. Most applications can be implemented with idempotent requests. The server can then trade the cost of storing for reprocessing, which is usually comparatively low. Sensor readings for a GET request, for instance, can be cached in a compact internal format and then serialized to the preferred representation. PUT requests only trigger an action if the request body differs from the current resource state. In the case of non-idempotent POST requests, the resource handler can leverage knowledge about the implementation to optimize the deduplication state. In its simplest form, deduplication state could only be stored for such particular resources. Compared to having deduplication state for all resources, the number of interactions with a server is less restricted.

REST Layer Abstraction

When we started the Erbium (Er) project, HTTP was still considered a reasonable choice for resource-constrained devices in the WoT. Hence, we adopted the REST Layer abstraction and provided a convenient way to bind either CoAP or HTTP as protocol. However, the interest in HTTP has been stagnating and there as been no effort to provide an HTTP binding. Yet Erbium itself has been adopted for several projects [5, 8, 17, 33, 38, 53, 54, 65, 88, 132, 134, 136, 150, 159, 177–179, 181]. Thus, we conclude that CoAP is the predominant protocol for the thin server architecture. The indirection of REST Layer abstraction can be removed to lower the memory footprint to 3–4 KiB, given that usually not all features of CoAP are used (e.g., only required options or no Link Format filtering).

3.4 Evaluation and Results

We use our Erbium reference implementation to evaluate the Constrained Application Protocol (CoAP) in constrained environments, that is, resource-constrained devices and low-power lossy network (LLN). We evaluate both static and dynamic properties of our low-power CoAP: memory footprint, energy consumption, and request round-trip delay times (RTTs), which denotes data throughput. Our evaluation is based on Contiki 2.4[16] and the code used is available on GitHub[17].

3.4.1 Experimental Setup

We run all our experiments on Tmote Sky sensor motes. The platform is based on a MSP430 16-bit CPU running at 3.9 MHz. It provides a CC2420 radio chip, 48 KiB of program flash and 10 KiB of RAM. We use a small, linear 4-hop network with static routes as depicted in Figure 3.5 to have a controlled topology. One Tmote Sky implements the 6LoWPAN border router connected to a computer running Linux. The IEEE 802.15.4 radio is configured to channel 15, which is affected by Wi-Fi interference of an office environment. Unless explicitly mentioned, we always use ContikiMAC and set the listener wake-up frequency to 8 Hz, which corresponds to a 0.6% idle duty cycle. The results displayed are averaged over 100 runs and error bars show the standard deviations. We characterize the device energy consumption using Contiki's energy profiler Powertrace [46].

[16] Git commit 78f7a746891c41af5515d2b5e01687a4461613ed

[17] https://github.com/mkovatsc/SmartAppContiki/tree/coap-06-bench-memory (accessed on 12 Feb 2015)

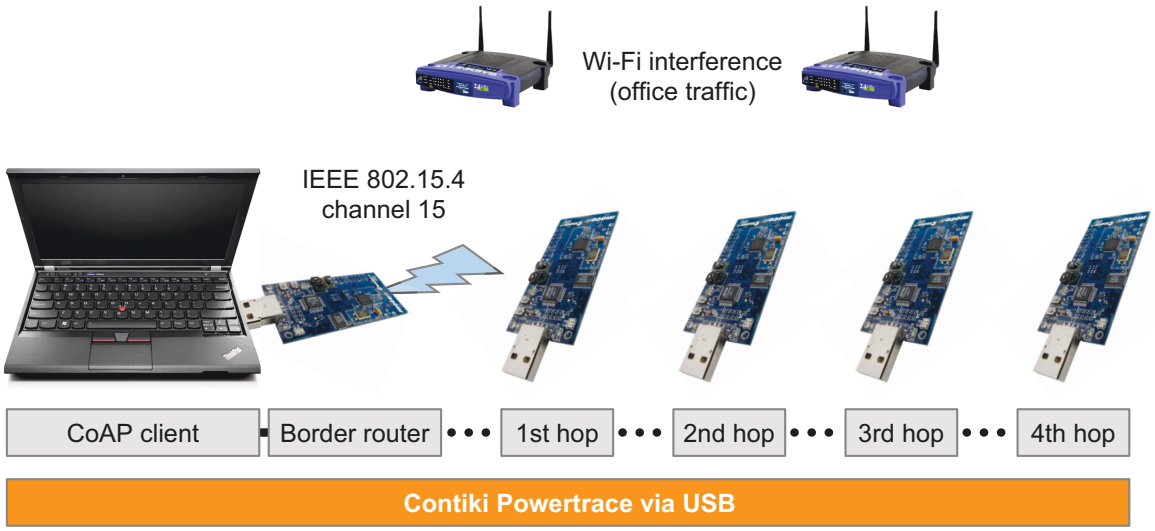
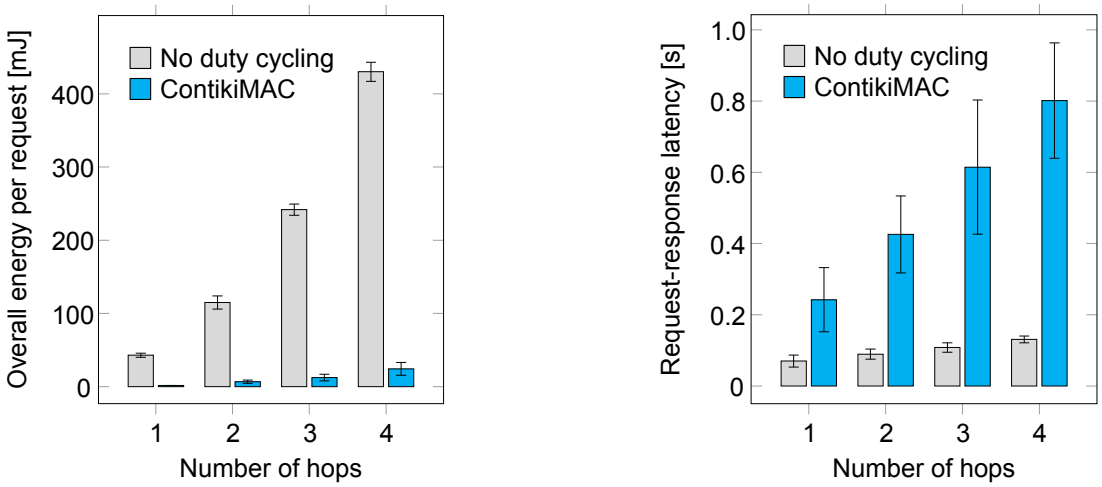


Figure 3.5: By disabling RPL, we can control the topology with static routes and also gain memory space for Powertrace.

3.4.2 Energy Consumption

We are not the first to argue the benefit of isolating low-power mechanisms to a single layer, but we are the first to demonstrate the implications of duty cycling for IoT application layers, as exemplified by CoAP [107]. To evaluate the trade-off between energy consumption and latency, we set up an experiment in which we issue requests to different motes in our 4-hop network: in one case without RDC (best case latency) and in the other with ContikiMac enabled (0.6% RDC). The requested CoAP resource responds with an echoed 2-byte token and a fixed payload of 64 bytes. Figure 3.6(a) shows the cumulated consumption of all devices involved in the process. This includes the targeted CoAP server as well as forwarding nodes. As expected, ContikiMAC saves a lot of energy, reaching an improvement by a factor of 26 compared to no duty cycling. Figure 3.6(b) shows that the energy savings are traded for latency. An RTT of less than one second for four hops is acceptable, though, considering the duty cycle of well below the 1% mark.[18] The maximum measured slow-down was about factor 6. We argue that the substantial lifetime increase offered by ContikiMAC is in many cases worth paying this latency overhead. As a more general result, this shows that existing application-layer protocols can be made energy-efficient through a separate, transparent RDC layer.

[18] Note that the actual duty cycle depends on the data traffic and noise on the channel.



(a) Energy

(b) Latency

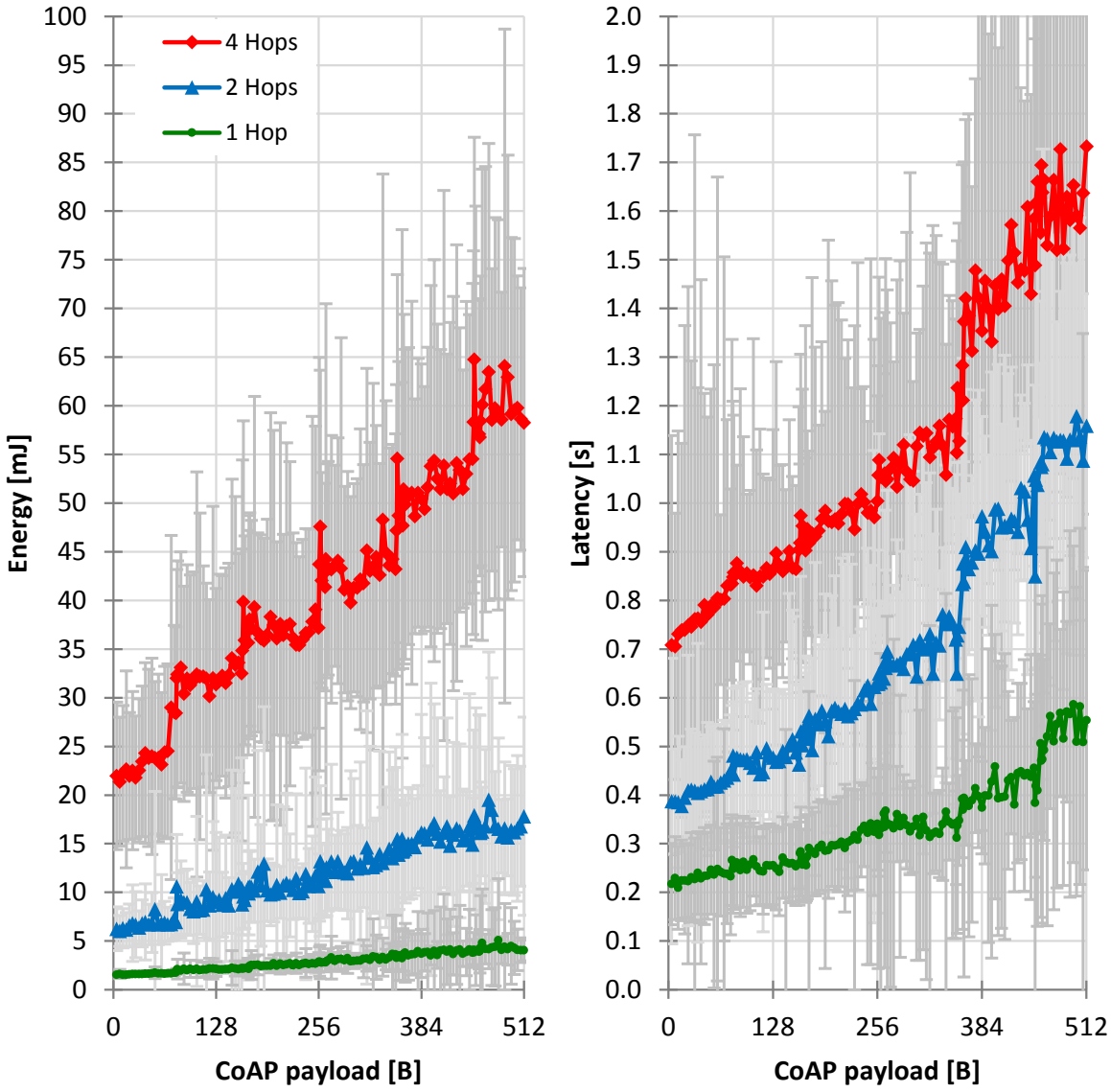
Figure 3.6: The overall energy consumption and latency for a request with 64 bytes payload in the response and no fragmentation. ContikiMAC substantially reduces the device energy consumption while keeping a reasonable end-to-end latency.

3.4.3 Transmitting Large Data

Not all CoAP resource representations can fit into a single 802.15.4 frame, so that either 6LoWPAN fragmentation or blockwise transfer is required. We focus on fragmentation to analyze the energy consumption as a fine-grained function of the payload size using RDC. The energy cost of blockwise transfers would simply correspond to multiple requests with the payload size adjusted for the additional `Block2` option. To enable energy-efficient transmission of consecutive frames, we also use link-layer bursts. Multiple frames are sent and acknowledged consecutively until the frame pending bit is unset.

We ran an experiment in which a client on the computer requests a resource of a CoAP server, which also echoes a 2-byte token in its response. The payload of the response ranges between 4 and 512 bytes with 4-byte increment steps. The client again targets motes at 1, 2, and 4 hops from the border router. We expected to see a step-wise increment of the energy consumption as the number of fragments increases.

Figure 3.7(a) shows the cumulative energy consumption of all nodes involved in the request (server and forwarding). The request-response latency is shown in Figure 3.7(b). The increase in energy consumption is clearly noticeable for the first additional fragment at around 69 bytes of payload. The following fragment steps slowly become less distinguishable. This is due to the large standard error caused by the varying link quality of low-power communications: The experiment was run in offices and the devices were



(a) Energy vs payload size

(b) Latency vs payload size

Figure 3.7: In lossy environments, the energy cost and latency when sending large CoAP payloads depends more on the payload size than the number of required 6LoWPAN fragments. This is reflected in the disappearing steps with increasing payload size. The reason for this is link-layer retransmission.

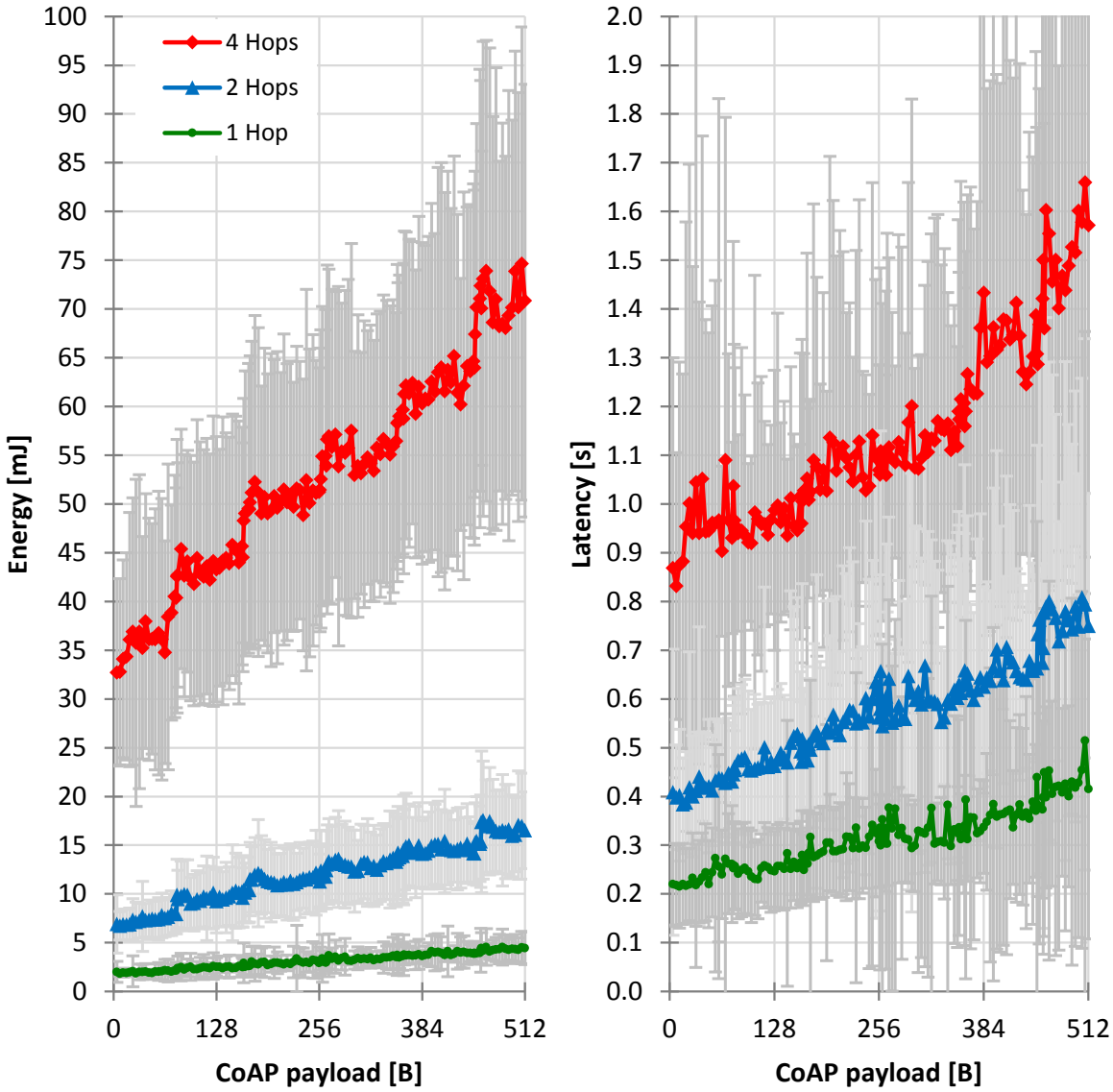
subject to Wi-Fi interference. When packet loss occurs, ContikiMAC simply transmits additional data frames until receiving an acknowledgment. This is simply an aggressive form of link-layer retransmissions with which ContikiMAC increases reliability in addition to lowering the energy consumption. Because of this lossy environment, the total number of frames sent depends more on the local link quality than on fragmentation. As a result, the total energy cost and latency are more closely proportional to the payload size and not discretely to the number of 6LoWPAN fragments.

This hypothesis was confirmed with an additional experiment that was conducted during the weekend. The overall trend of Figure 3.8 is similar to Figure 3.7. The more or less constant standard deviation in Figure 3.8(a) indicates are more stable link quality in the office environment. Yet the channel is more lossy. On average, the energy consumption for four hops is about 30% higher. In summary, this evaluation shows that the exact number of used link-layer frames is insignificant for the overall performance.

We used the more stable weekend office environment to analyze the impact of the link-layer bursts in more detail. Figure 3.9(a) shows again the latency for 1 and 2 hops. Next to it, Figure 3.9(b) shows the latency when link-layer bursts are disabled. The RTT of the request-response exchanges is more than doubled. For one fragment, the curves are similar, as the message has to wait roughly one channel-check interval per hop and direction. For two fragments, sending devices have to wait one additional channel-check interval for the second fragment. The blue curve (triangular markers) also has data points on the flanks between the steps. These result from changes in the IPv6 header along the routing path: One hop from the border router, the header is shorter. Thus, no fragmentation is required for one of the two hops. This results in a latency of about five channel-check interval (two for delivering the request, two for the two response frames from hop 2 to hop 1, and one for the unfragmented response from hop 1 to the border router). This behavior also explains the intermediary data points on the flanks of the energy cost of Figure 3.7(a) and Figure 3.8(a). The main take-away from this experiment is that link-layer bursts significantly lower the latency, and hence link-layer fragmentation can be favorable over application-layer fragmentation (i.e., blockwise transfers), which cannot benefit from this technique.

3.4.4 Memory Footprint Optimization

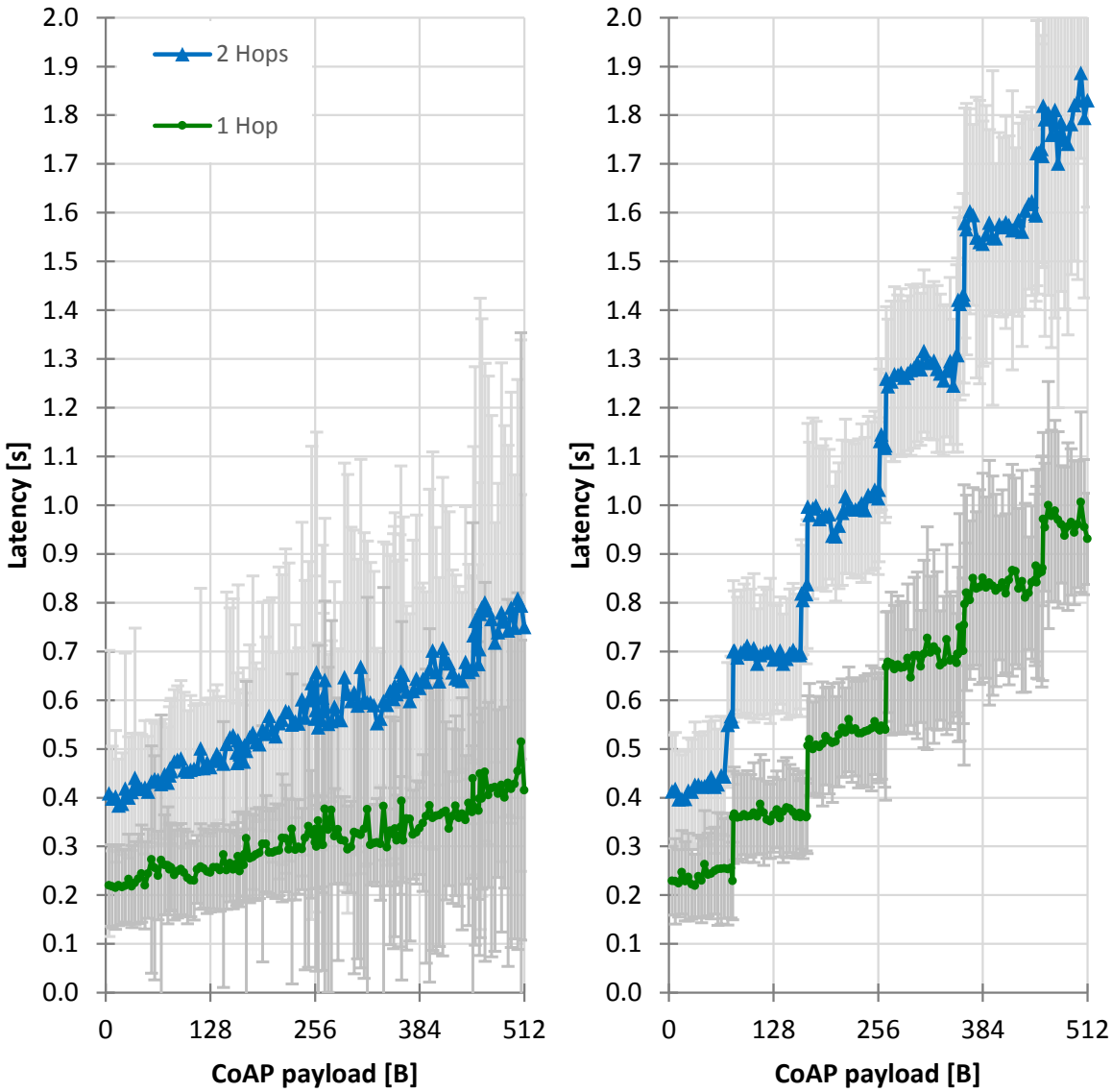
We also analyze the ROM and RAM requirements of our RFC7252-compliant Erbium (Er) REST Engine. As a reference, we use two typical low-cost IoT platforms: TI's MSP430 and Atmel's 8-bit AVR. The code is compiled using the *msp430-gcc* and *avr-gcc* compilers in version 4.5.3 with the default compiler flags of the Contiki build system.



(a) Energy vs payload size

(b) Latency vs payload size

Figure 3.8: With a more stable link quality, the fragmentation steps are slightly more prominent. The link quality is worse, resulting in a higher standard deviation and higher energy baseline. The curves also becomes closer to a linear dependency on the payload size.



(a) Latency with link-layer bursts

(b) Latency without link-layer bursts

Figure 3.9: Link-layer bursts significantly lower the RTTs of multi-hop requests. The data points on the flanks between the steps of the 2-hop curve in b) result from changes in the IPv6 header along the routing path.

When including all CoAP features and convenience functions, the ROM footprint is 9.5 KiB and 13.5 KiB, respectively. Besides static RAM usage given by the object code, we measured the stack while handling a CoAP request. For this, we observed the stack pointer and logged the growth above the stack size in the scheduler function of the OS. In total, Erbium requires about 1.5 KiB of RAM when supporting four retransmission buffers with a maximum chunk size of 128 bytes. Table 3.1 shows the breakdown into the different C modules.

The business logic on top of Erbium is compact, as most features are already provided by the REST Engine API. A thin server with five resources as depicted in Table 3.2 (in addition to the `/.well-known/core` resource) only consumes 1.5 KiB of ROM and 0.2 KiB of static RAM. Overall, Erbium's memory footprint is much smaller than comparable CoAP implementations such as *libcoap*, which requires up to 80 KiB of ROM when compiling it with similar Web resources. The memory usage can be further optimized by removing unused modules or the REST Layer abstraction with its convenience functions.

3.5 Summary and Discussion

In this chapter, we focused on the research question how Web technology can be scaled down to fit tiny, resource-constrained devices and low-power networks. We showed how to embed Web servers with the rich features of REST on tiny resource-constrained devices.

Memory footprint [KiB]	**MSP430**		**AVR**	
	ROM	*RAM*	*ROM*	*RAM*
Erbium (Er) REST Engine total	9.46	1.47	13.50	1.72
`rest-engine`	0.62	0.01	0.83	0.03
`er-coap`	4.63	0.01	6.62	0.14
`er-coap-engine`	1.62	0.27	2.05	0.35
`er-coap-transactions`	0.39	0.85	0.50	0.83
`er-coap-observe`	0.96	0.17	1.43	0.16
`er-coap-separate`	0.33	0.00	0.51	0.01
`er-coap-block`	0.25	0.00	0.40	0.02
`er-coap-res-well-known-core`	0.66	0.02	1.16	0.04
Measured stack usage	–	0.14	–	0.14

Table 3.1: The AVR program memory footprint is always slightly larger than that of the MSP430. Due to the modified Harvard architecture of AVR, however, RAM usage can be optimized by defining strings in the program memory (not applied in the Erbium code).

URI-Path	*Functionality*
`/battery`	Voltage level as text or JSON
`/event`	Button press notifications
`/leds`	Control via key-value pairs in query and payload
`/light`	Both light sensor readings as text or JSON
`/push`	Periodic notifications

Table 3.2: Our measured example application provides five non-trivial resources with Link Format descriptions between 32 and 85 bytes. As Erbium provides most functionality for RESTful Web services, typically the business logic only uses 1–2 KiB of ROM.

Section 3.2 introduced our *thin server architecture*. We propose to separate the application logic from the embedded device firmware with the help of CoAP. On the one hand, this enables an application-agnostic infrastructure of IoT devices that provide their elementary hardware functions through RESTful interfaces. The application logic can then run outside the embedded domain on any machine with network connectivity: on a more powerful device directly in the LLN, the border router, a local server, or the cloud. Multiple applications can leverage the device infrastructure in parallel by mashing up their CoAP resource. The uniform REST interfaces also enable an open market place where IoT services and apps can be provided by third parties. On the other hand, the thin server architecture eases application development for the IoT. By keeping the device firmware free of application logic, it is easier to maintain. Networked embedded systems experts are scarce and a high code quality is of importance to ensure a high security standard. Moreover, the separation of application logic enables developers with different backgrounds to create innovative applications for the IoT and contribute to the new economy.

With the *Erbium (Er) REST Engine* for resource-constrained devices, we provide an implementation of the thin server architecture. We confirmed our hypotheses on the applicability of Web technology in resource-constrained environments through the realization of working IoT device prototypes. Examples are given in Figure 3.10. Furthermore, we used this CoAP implementation to evaluate the design decisions in the standardization process and inferred implementation guidelines. Our five Web resource types, `RESOURCE`, `PARENT_RESOURCE`, `EVENT_RESOURCE`, `PERIODIC_RESOURCE`, and `SEPARATE_RESOURCE`, cover the design patterns for CoAP-based applications. With these abstractions, the implementation of thin servers also looks similar to classic REST frameworks that are used in the Web. To simultaneously achieve a small memory footprint, we primarily make use of in-place processing and the reuse of buffers. Other optimizations address the observe feature, blockwise transfers, and message deduplication.

Finally, we experimentally evaluated CoAP in a realistic low-power setting: a 0.6% idle RDC to achieve a high energy efficiency, link-layer bursts for fragmented packets, and IP multihop routing on an IEEE 802.15.4 channel affected by office Wi-Fi traffic. We showed that the use of a generic duty cycle mechanism is transparent to the application layer and results in a low power consumption—at the cost of a higher latency. Our experiments confirm that CoAP request-response exchanges are most energy-efficient when each message fits into a single IEEE 802.15.4 frame. Once fragmentation is performed, however, there is no need to optimize the number of fragments. The number of transmitted frames is dominated by the link quality and clock synchronization, which affects the length of the RDC strobe. Consequently, an optimization of the CoAP block size definitions for 6LoWPAN fragments has no significant benefit, at least when link-layer bursts and a sender-initiated RDC layer are used. Furthermore, 6LoWPAN fragmentation and blockwise transfers should be used in combination: The block size should be chosen so that it is large enough for all responses of the sensing and actuation resources (usually 128 or 256 bytes). Then latency can be optimized significantly through link-layer bursts. Large resources such as `/.well-known/core` should resort to blockwise transfers instead of heavy 6LoWPAN fragmentation. The evaluation of Erbium's memory footprint shows that the REST architectural style of the Web can be realized efficiently for tiny, resource-constrained devices.

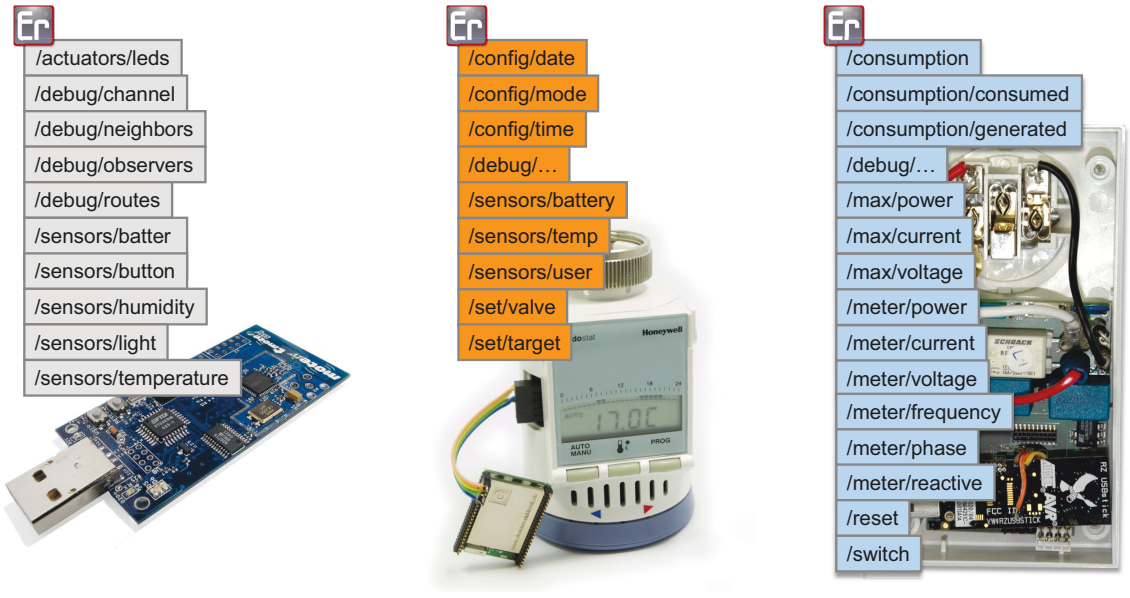


Figure 3.10: Our thin server prototypes include the Tmote Sky sensor mote, a smart thermostat, and a smart power outlet. The thermostats, for instance, have been deployed in our offices at ETH Zurich and a test household in Germany for a long-term deployment in a Smart Energy study.

The thin server architecture also introduces new challenges. An open question, for instance, is where exactly to make the cut between device functionality and application logic. This is particularly interesting for cyber-physical systems that have hard real-time requirements. Often, these cannot be met because of the RTTs to a cloud service. Thus, these control loops must be realized closer to the devices. The question where to cut and execute is out of scope of this thesis, though. It is tackled by the recently defined research field of fog computing [21, 185]. One aspect that we do address in Chapter 5, however, is a suitable runtime system for the IoT application logic. A related issue is in-network processing, which was, for instance, raised by the T-Res project [7]. When all data is transferred to the cloud and decisions are carried out by sending commands back to the devices, the border router or even the LLN could become congested. Since resource-constrained devices and application services have the same RESTful interfaces, though, the application logic can run anywhere. More powerful devices in the LLN could perform in-network processing similar to cluster heads in traditional WSN approaches [60, 75, 85]. A third challenge is safety, which must be ensured independently from fluctuating network link quality. For any API input a device must enforce local safety rules, for instance, to avoid hazardous valve or motor control. The connected security and privacy challenges are independent from the thin server model, as the requirements apply to any architecture. CoAP addresses the latter through Datagram Transport Layer Security (DTLS) and per-resource authentication and authorization recently picked up by the Authentication and Authorization for Constrained Environments (ACE) working group. A more concrete issue is the lack of an open-standards-based RDC protocol for low-power IP stacks. At the time of writing, this is one of the main show-stoppers for interoperable IoT devices in the real world. It also also the reason why industry still prefers the sleepy node approach. The 6LoWPAN binding for BLE could become a solution for an energy-efficient Web of Things in consumer electronics. In the industrial area, the effort by the IETF 6TiSCH working group appears to be a good candidate for a standardized RDC protocol that also provides deterministic transmission of messages.

Chapter 4

IoT Cloud Services and Scalability

The Internet of Things (IoT) is expected to interconnect vast numbers of devices. Given the low unit costs and open Internet standards, analysts expect up to 200 billion connected devices by the end of 2020 [3, 125]. Furthermore, the generated traffic is quite different from human-centered Web applications, multimedia streaming, and file sharing. Instead of bulk data, IoT nodes will primarily exchange real-time sensory and control data in small but numerous messages. Thus, emerging networking and backend support technology not only has to anticipate a dramatic increase in connected nodes, but also a change in traffic patterns.

While devices will be deployed in many different environments of interest, such as homes, office buildings, factories, and whole cities, a common trend is using cloud services to manage large numbers of devices, process their data, and orchestrate their actuation. To this end, we present a scalable system architecture for IoT cloud services that allows to build conceivable large-scale IoT applications. Furthermore, we systematically evaluate the performance of CoAP in cloud environments. The results show that CoAP-based backend systems outperform classical high-performance HTTP Web servers, which today are the state of the art for cloud services. Our implementation, the Californium (Cf) *CoAP framework*, has 33 to 64 times higher throughput and scales well for vast numbers of concurrent clients. The results substantiate that the low overhead of CoAP does not only enable Web technology for low-cost IoT devices, but also significantly improves scalability for IoT cloud services.

This chapter is based on our publications [109] and [110]. We first give an introduction to related work, before we present our system architecture in Section 4.2. Section 4.3 gives an overview of our reference implementation, which is used in Section 4.4 to evaluate the scalability of our approach.

4.1 Related Work

Traditional networked embedded systems and WSNs usually use a so-called sink node that connects them directly to a local service host [97, 128]. This can also involve long-distance communication via microwave links or cellular networks between sink and host [18, 195], but usually no routing through the Internet. Since the service host is dedicated to a single deployment, no sophisticated system architectures have been necessary and little research has been conducted on the scalability of services.

4.1.1 From Sink Nodes to Web-based Services

While single sensor deployments can help domain experts to answer specific questions, their measurements can become even more valuable when combined with complementary data from the same region, related data from other regions, or similar data from different periods in time. This trend is commonly called *big data* and requires broader access to the deployments and their data. For this, the dedicated service hosts have been upgraded to *application-level gateways* that connect the WSNs to syndication platforms. The gateways therefore must be able to translate the application-specific messages into platform-compatible representations, usually common Web protocols and formats. The *Global Sensor Network (GSN)* middleware [6] uses the concept of virtual sensors to abstract from the heterogeneous hard- and software. Furthermore, it provides a Web interface for services and Web-based management tools for users, both based on standard HTTP libraries. A comparable approach is implemented by *sMAP*, the Simple Measurement and Actuation Profile for Physical Information [41]. It uses RESTful interfaces and a JSON-based object model to exchange data from heterogeneous sensor and actuator networks. For 6LoWPAN-based networks, it uses EBHTTP [180] (see Section 3.5) and Apache Avro[1] as binary representation of the JSON objects. However, sMAP gateways also allow for the integration of other networked embedded systems such as Modbus, BACnet, and Supervisory Control and Data Acquisition (SCADA). The *SenseWeb* project by Microsoft Research [101] was one of the first IoT cloud services. It uses stream processing to aggregate data from devices and other sources and presents them through a map-based Web application. Data objects must be added using the .NET framework, though. Further developments, such as the commercial *Xively* platform[2] (formerly known as *pachube* and then *Cosm*) are fully Web-based and use a RESTful API to feed sensor data into the system.

[1] http://avro.apache.org/ (accessed on 12 Feb 2015)
[2] https://xively.com/ (accessed on 12 Feb 2015)

Most service-related projects rely on HTTP libraries to scale with the number of connected gateways and users. This is a sensible design decision because HTTP Web servers implement the state of the art in scalable system design. Our goal is to enable Web technology directly for resource-constrained devices, though. Thus, we propose a CoAP-based system architecture for IoT cloud services. To understand our system design, we first give a detailed overview over the state of the art in Web server architectures, before we present related CoAP frameworks in Section 4.1.3.

4.1.2 Web Server Architectures

Architectures for Web servers have been enhanced since the advent of the Web itself to scale with the ever-growing traffic. The central strategy for scalability is concurrency, the parallel execution of tasks, especially since CPU clock rates peaked in 2004 due to power consumption and heat dissipation, and modern computing hardware becomes more powerful by integrating multiple cores instead. For Web servers, the concurrent handling of requests was originally introduced to accept connections form multiple clients at the same time to better utilize the (single-core) CPU during I/O operations. Since the introduction of HTTP/1.1 in 1999, clients have also been allowed to keep connections alive for multiple consecutive requests [69]. The goal is to reduce round-trip delay time (RTT) by avoiding the 3-way handshake and TCP slow-start mechanism for every new request, since a typical Web page contains several links to images and other content that needs to be loaded. In the past 15 years, different server architectures evolved, which all attempt to increase server efficiency by mitigating bottlenecks such as synchronization overhead. In the following, we present the relevant literature as well as popular projects that implement the different architectures:

1. Multi-Process (MP)
2. Multi-Threaded (MT)
3. Single-Process Event-Driven (SPED)
4. Asynchronous Multi-Process Event-Driven (AMPED)
5. Staged Event-Driven Architecure (SEDA)
6. Multi-Threaded Pipelined (PIPELINED)

Table 4.1 summarizes the procedure to handle an incoming request by a Web server. The left column shows the major steps necessary for classic HTTP servers. Originally, HTTP servers served documents that are stored on disk and only need to be read and sent as stream over the TCP connection. This still holds for so-called assets on Web pages such as images or stylesheets. Nowadays, there are also many dynamic Web resources, which usually process a file that contains static content as well as code instructions to

	HTTP	CoAP
1	Accept connection	Receive datagram
2	Interpret the request	Interpret the request
3	Find file/resource handler	Find resource handler
4	Send the response header	—
5	Read file/execute handler and send stream	Execute handler and send datagram

Table 4.1: Simplified processing steps of a Web server to handle requests.

dynamically generate the responses (e.g., PHP, ASP, JSP). By contrast, CoAP and the IoT are mainly about real-time data and control commands. Thus, in most cases there is no classic file access. Yet the resource handler might trigger a script or database query that is currently not cached in memory. The right column of Table 4.1 shows the required processing steps for CoAP Web servers. Besides the fact that the headers are sent together with the response body in a single message, the procedures are similar for both protocols.

Multi-Process

The first step toward parallelization in Web servers was the Multi-Process (MP) architecture, which dates back to the original *CERN httpd* project[3]. The workload is distributed over multiple processes as depicted in Figure 4.1. When a client establishes a new TCP connection, the server forks a new process to handle the incoming requests. This is supported by most operating systems along with sophisticated scheduling strategies for optimal CPU usage fairness. Multiple processes also provide higher reliability, as they are handled independently by the OS and faults can be isolated. Subsequent implementations of this architecture often use pre-forked worker processes to improve performance. An example for this is the *NCSA HTTPd* server, whose codebase was later adopted by the initial version of the prevalent *Apache HTTP Server*[4]. The current version 2.4 still uses this architecture in its default Multi-Process-Module (`mpm_prefork_module`), the component that handles network I/O, to support non-thread-safe libraries such as the popular PHP module (`mod_php`). A drawback of MP is the overhead, however, when sharing global information such as a document cache. MP servers also lack sufficient management of machine resources for priority policies and robust implementation of services. [10]

[3] `http://www.w3.org/Daemon/` (accessed on 12 Feb 2015)
[4] `http://httpd.apache.org/` (accessed on 12 Feb 2015)

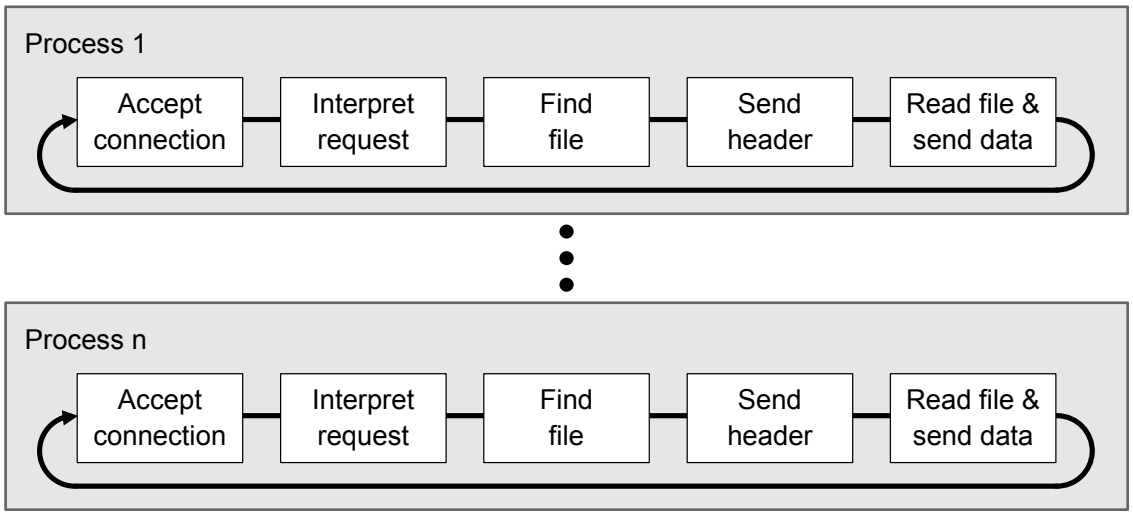


Figure 4.1: Multi-Process architecture: Each process executes the protocol steps synchronously and blocks on I/O operations.

Multi-Threaded

The Multi-Threaded (MT) architecture (see Figure 4.2) uses threads instead of processes, which reduces the overhead of forking. Threads are also able to access a shared memory and there exist many libraries for efficient caching. This is supported by modern programming languages, which provide powerful synchronization mechanism for multi-threading. It is crucial, though, that the operating system used supports kernel-threads to make use of multiple cores. Usually, user-level threads only support cooperative scheduling, as they are managed by the application. Furthermore, the whole process may be slowed down during blocking I/O operations, since the interrupt handler of the kernel cannot preempt user-level threads to schedule another server thread.

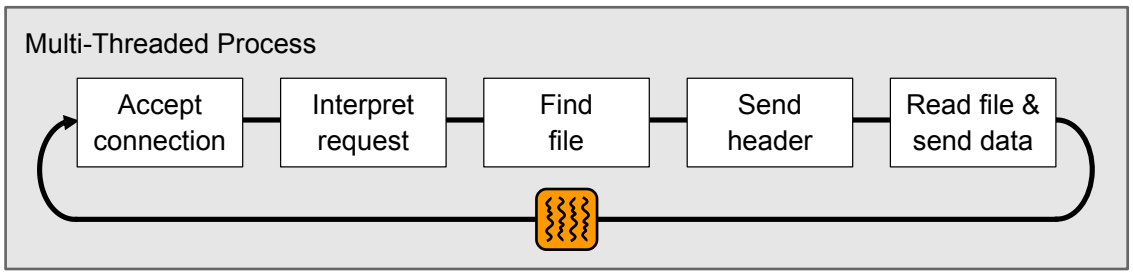


Figure 4.2: Multi-Threaded architecture: Executing the protocol steps works similar to MP, but with reduced overhead for additional workers.

In a typical multi-threaded HTTP server, a welcome socket accepts new TCP connections, creates a new socket and assigns it to a thread. This results in one thread per connection and allows to serve each request without delays. Best performance is achieved when the number of threads equals the number of expected concurrent requests [15]. Similar to the pre-forked processes, most implementations use a pool of reusable worker threads. However, threads still allocate a considerable amount of memory for their state, in particular the separate stacks. On a 64-bit Java Virtual Machine, for instance, the default stack size of a thread is 1 MiB.[5] In addition, the synchronization of a large number of threads can lead to a significant overhead. Thus, the maximum number of threads is usually limited.

MT has been widely used since the end of the 1990s and is implemented by Apache's Multi-Process-Module for Windows (`mpm_winnt_module`) and by an alternative Multi-Process-Module for Linux (`mpm_worker_module`), which actually combines MT with MP by having multiple child processes with multiple worker threads. *Tomcat*[6], a Web server for Java Servlets and JavaServer Pages (JSP), uses this architecture as well, although it allows for the optional configuration of alternative network connectors that implement newer architectures. This is because MT and MP alike do not perform well for high numbers of concurrent clients. When all threads are assigned, no further connections can be accepted until a connection closes. This is a problem for modern Web applications with many users that use keep-alive connections and WebSockets [68] for push notifications from the server to the client. Thus, all following architectures were specifically designed to overcome this limitation.

Single-Process Event-Driven

The Single-Process Event-Driven (SPED) architecture of modern high-performance Web servers leverages the same technique as operating systems for resource-constrained embedded systems: an event-driven approach with non-blocking I/O operations. One of the first Web systems using this architecture was the *Harvest* object cache by Bowman et al. [28], which later became the *Squid*[7] proxy server. SPED splits the request handling procedure up into small tasks that result from certain events such as a new inbound connection, completion of a file operation, or free space in the send buffer of a response stream. They are managed in a global event queue and a single thread executes them one after the other as shown in Figure 4.3. Despite having only one thread, SPED is able to parallelize the usage of the different computing resources such as CPU, disk, and network. Since

[5] `http://www.oracle.com/technetwork/java/hotspotfaq-138619.html` (accessed on 12 Feb 2015)
[6] `http://tomcat.apache.org/` (accessed on 12 Feb 2015)
[7] `http://www.squid-cache.org/` (accessed on 12 Feb 2015)

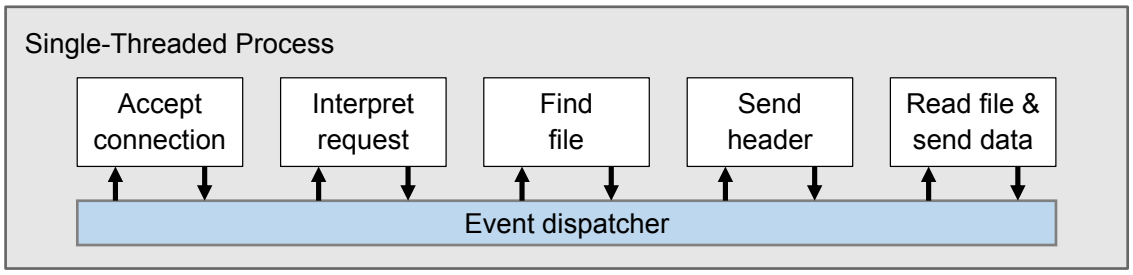


Figure 4.3: Single-Process Event-Driven architecture: A single thread jumps from task to task and uses non-blocking I/O operations.

no synchronization is required for a single thread, context switches can be saved and data are always cache-local. *Node.js*[8] is a good example for SPED's efficiency. It is based on Google's V8 JavaScript Engine[9] and introduces server-side JavaScript (a.k.a. ECMAScript [57]) for cloud services. The scripting language is event-driven itself, since it was originally developed to handle input events for dynamic Hypertext Markup Language (HTML), and usually runs with a single thread.

The disadvantage of SPED is that a single thread cannot benefit from multiple cores and, unfortunately, many operating systems do not provide suitable support for non-blocking operations [146]. Thus, Pariag et al. proposed an extension called Symmetric Multi-Processor Event-Driven (SYMPED), which forks multiple SPED processes [149]. This architecture also solves the problem of SPED processes getting stuck in blocking I/O operation, since the OS can then switch to another server process that is able to run.

Asynchronous Multi-Process Event-Driven

A more fundamental extension of SPED is the AMPED architecture, which was published with the *Flash* Web server [146]. It also uses a single dispatching thread like shown in Figure 4.4. This process only serves requests with a cache hit, though. When there is a miss, the dispatcher forwards the request to a helper process (or thread) to fetch the data, for instance from the disk. In other terms, AMPED wraps blocking I/O operations in separate processes (or threads) to turn them into asynchronous, non-blocking operations. The Flash Web server by Vivek et al. [146] was originally designed for the single-core systems of the late 1990s. In 2000, Palchaudhuri et al. hence designed the Co-AMPED architecture for multi-processors. It uses one AMPED process per core and outperforms Apache's MP implementation [147].

[8] http://nodejs.org/ (accessed on 12 Feb 2015)
[9] https://developers.google.com/v8/ (accessed on 12 Feb 2015)

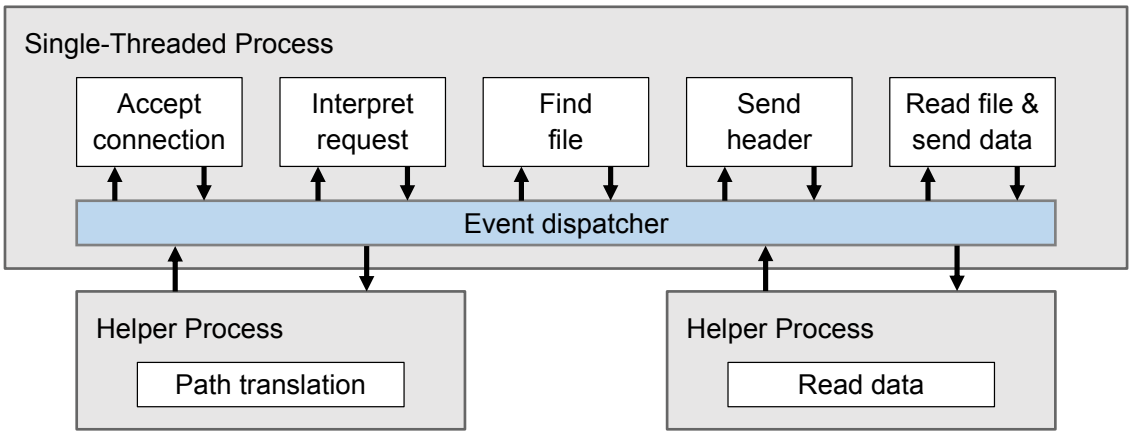


Figure 4.4: Asynchronous Multi-Process Event-Driven architecture: Outsources blocking operations to separate processes.

Today, most high-performance Web servers are based on non-blocking I/O and use an event-driven architecture for multiple cores. After J2SE 1.4 introduced the *New I/O API* (NIO) in 2002, Beltran et al. showed that an event-driven Web server written in Java and using NIO "[...] scales as well as the best of the commercial native-compiled Web server, at a fraction of its complexity and using only one or two worker threads" [14]. This is exemplified by project *Grizzly*[10], which started as a component of the GlassFish Java Enterprise Edition application server to provide a more scalable HTTP server interface. A comparable project that has become popular for Web-based M2M is *Jetty*[11], which also provides a Java Servlet container. Node.js is widespread for IoT services as well. To utilize multi-core systems for better scalability, it provides the so-called *cluster mode* where one Node.js process is started per core, similiar to Co-AMPED. A recent project that closely represents the Co-AMPED architecture is the polyglot *Vert.x* Web application platform. It combines non-blocking network I/O (using the *Netty* project[12]) with support for several languages to implement the business logic (e.g., Java, JavaScript, Ruby, and Python). The latter is done in so-called *Verticles*, which package application code and are executed by a single thread. Usually, there are multiple instances of the same Verticle, for instance one per core. As proposed by AMPED, blocking operations are moved to special *Worker Verticles*.

[10] https://grizzly.java.net/ (accessed on 12 Feb 2015)
[11] http://www.eclipse.org/jetty/ (accessed on 12 Feb 2015)
[12] http://netty.io/ (accessed on 12 Feb 2015)

Staged Event-Driven Architecture

The Staged Event-Driven Architecure (SEDA) [192] combines the event-driven approach with multi-threading by splitting message handling into multiple stages as shown in Figure 4.5. Each stage is responsible for a specific task, such as interpreting the request or loading a file from disk, and consists of an incoming event queue, a thread pool, and an event handler that executes the logic of the stage. The threads pull events from their incoming event queue and invoke the event handler, which can dispatch new events to the next stage through their connecting queue. Each stage is managed by a controller that can dynamically change the configuration according to a policy. Therefore, each stage of a SEDA server can self-tune itself to have an optimal number of threads.

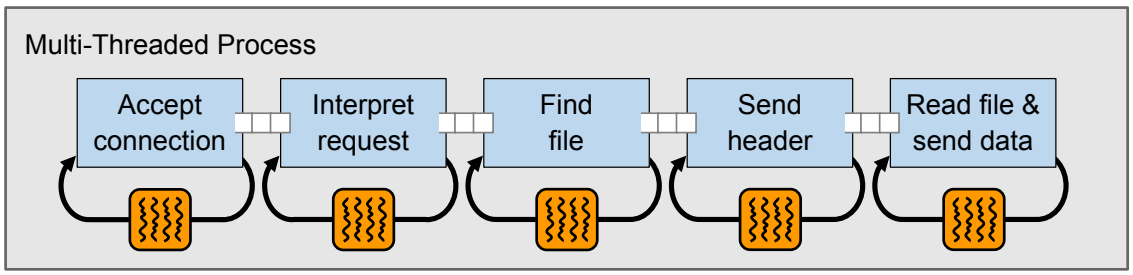


Figure 4.5: Staged Event-Driven Architecture: Each protocol step is a stage. Each has its own thread-pool and forwards processed messages to the next stage over a queue.

Welsh et al. implemented an HTTP server called *Haboob* based on SEDA, which outperforms MP Apache and AMPED Flash [192]. In 2002, Larus et al. showed that a SEDA server can even further increase its throughput when threads pull a batch of events from the event queue to improve cache-locality [114]. However, in 2003, von Behren et al. suggested to use compiler support to improve synchronization and memory stack management of MT servers and presented results that outperform Haboob once again [188].

Multi-Threaded Pipelined

A chain of queue-connected stages can be seen as a pipeline. In contrast to SEDA, the Multi-Threaded Pipelined (PIPELINED) architecture by Choi et al. [37] only has one thread per stage, but it creates one pipeline per core as shown in Figure 4.6. The advantage of this architecture is that all threads of one pipeline belong to the same pool so that the processing data is always cache-local. Additionally, a pipeline can use helper threads

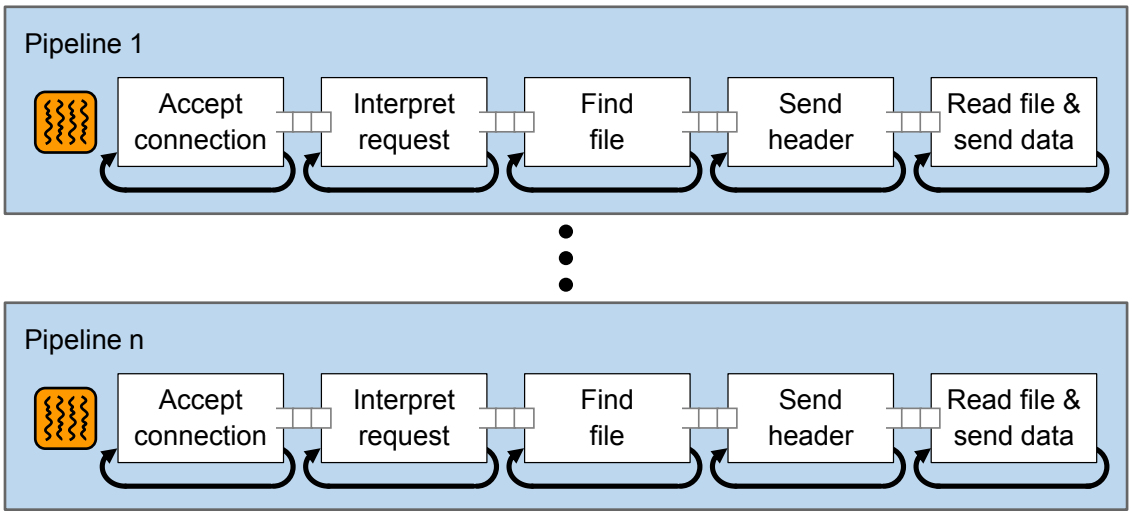


Figure 4.6: Pipelined architecture: The protocol steps form a pipeline where each step is processed by one thread. Blocking operations can be outsourced to additional threads (not in the image).

to outsource blocking I/O in case of document cache misses, similar to AMPED. It was shown that PIPELINED outperforms MP, MT, SPED, and AMPED in terms of memory usage and throughput [37]. In 2007, Pariag et al. confirmed that both PIPELINED and Symmetric Multi-Processor Event-Driven (SYMPED) can achieve an 18% higher throughput than MT [149].

Comparison

An advantage of event-driven architectures over MT and MP is that fewer threads are necessary to handle a large amount of parallel connections, resulting in a lower synchronization overhead. Beltran et al. showed that a hybrid Tomcat server [34] that combines MT and SPED is able to outperform MT Tomcat on CPU-bound workloads when the number of concurrent clients exceeds 2000, while performing equally up to this point [15]. Many available implementations are in fact hybrids and combine approaches from the different fundamental architectures.

SEDA and PIPELINED both scale better than MT, but still suffer from synchronization overhead between each stage. Because of the connecting queues, the time a request spends in the system can significantly increase due to costly context switches and the potential loss of cache-locality. In 2010, Matt Welsh, the creator of SEDA, reviewed his design and wrote in his blog that "Most stages should be connected via direct function call" [13].

[13] http://matt-welsh.blogspot.ch/2010/07/retrospective-on-seda.html (accessed on 12 Feb 2015)

In summary, there exist a plethora of architectures and implementations, but also as many experiments whose results heavily depend on the underlying platform (e.g., CPU, memory, and operating system) and workloads used (e.g., disk-bound, memory-bound, computation-bound, small/large files, and cache-hit rate). Thus, results of different papers are sometimes contradictory. In 2012, Harji et al. extensively benchmarked multiple Web servers on quad-core SMP systems and concluded "[...] that implementation and tuning of Web servers is perhaps more important than server architectures" [83]. This means, that a good architecture also requires a good implementation to perform to its expectations. For our work, we take the fundamental design choices of these architectures into account and compare our system to the production quality implementations of the presented Web servers using the same platform and workload (see Section 4.4).

4.1.3 CoAP Service Frameworks

Literature on CoAP-based service backends and their architectures is still scarce, since most studies have targeted resource-constrained environments [35, 40, 107, 111]. In the following, we present related work that provides CoAP implementations that focus on providing backend services while running in unconstrained environments, e.g., the cloud.

The *Sensinode NanoService Platform* [112] is a commercial solution that offers good support for industry-relevant features such as OMA Lightweight M2M (LWM2M) support and in-memory data grid caching for big data. It is written in Java and based on the SPED architecture using Java's non-blocking NIO API. At the time of writing, the NanoService Platform is providing cloud services for several commercial CoAP-based deployments. Its core CoAP protocol engine is also going to be included in the OpenJDK as part of a collaboration between Oracle and ARM, the company that acquired the Sensinode start-up, to "stimulate broad adoption of the CoAP protocol."[14]

An early open-source project also written in Java is *jCoAP*[15] by the University of Rostock. In particular, the authors provide one of the first CoAP-HTTP cross-proxies to connect CoAP-based devices to classic HTTP-based Web services [116]. The paper also shows experimental proof that RESTful caching helps to unburden resource-constrained servers and networks from too many requests from the Internet. jCoAP also uses the SPED architecture around the NIO API with a single worker thread that polls the NIO datagram channel, handles requests, and sends the resulting responses. At the time of writing, jCoAP only supports an earlier draft version of CoAP. Thus, it cannot be included in a direct comparison for the scalability evaluation in Section 4.4.

[14] https://blogs.oracle.com/henrik/entry/armtechcon2013 (accessed on 12 Feb 2015)
[15] https://code.google.com/p/jcoap/ (accessed on 12 Feb 2015)

nCoap[16] is a Java open-source project by the University of Luebeck. Similar to Vert.x, it is built on top of the event-driven Netty framework, which also provides non-blocking UDP channels. nCoAP instantiates twice as many threads as available cores and dispatches half of them as network worker threads, while the other half serves other I/O operations in the server. Since the Java Virtual Machine (JVM) is able to distribute threads over multiple cores (by using the `java.util.concurrent` package), this resembles the Co-AMPED architecture.

Java is also available on stronger embedded devices that can act as local service hosts, for instance the Raspberry Pi, the Intel Galileo, or middle-class smartphones. This field is targeted by *mjCoAP* [39], a light-weight CoAP implementation without dependencies on other projects. It uses the SPED architecture and is organized in three layers, one for messaging, one for reliable transmissions, and one for transactions. The messaging layer has a single thread that calls listener objects and provides non-blocking send operation. Developers implement the listeners, which can react to specific CoAP messages through filtering.

There are also projects in the popular Python language such as the *txThings* project[17]. It implements CoAP for the Twisted framework[18], which is a Python implementation of the reactor pattern [166]. This means that txThings is build around a SPED networking engine like most of the other projects. The way Web resources are implemented and the server is started is quite similar to our solution. Developers simply extend a `CoAPResource` base class and add them to the root resource of the server or build hierarchical resource trees.

OpenWSN [190] is a comprehensive IoT project at UC Berkeley.[19] Its main aspect is high reliability for low-power communication based on time slotted channel hopping (TSCH), which was developed within WirelessHART and standardized as IEEE 802.15.4e. Besides a full software stack for sensor nodes, OpenWSN offers a CoAP Python library[20] to implement backend services. It primarily targets easy interaction with OpenWSN devices, though. Following SPED, it uses a single receiver thread that dispatches incoming messages directly through a callback, which in turn sends the reply in the same thread. We include the CoAP Python Library in our evaluation in Section 4.4. Since it was never designed for scalability, however, it is benchmarked non-competitively.

[16] https://github.com/okleine/nCoAP
[17] https://github.com/siskin/txThings (accessed on 12 Feb 2015)
[18] https://twistedmatrix.com/ (accessed on 12 Feb 2015)
[19] http://www.openwsn.org/ (accessed on 12 Feb 2015)
[20] https://github.com/openwsn-berkeley/coap (accessed on 12 Feb 2015)

4.2 The Californium Architecture

Early on in our work on Web technology for resource-constrained IoT systems, we identified the need for a service counterpart running in unconstrained environments such as a local service host or the cloud. For this, we created the *Californium* project. It started as student lab project in 2011 [142] and after further development became 'running code' for the design and standardization of CoAP within the IETF. Since then, it has been one of the most comprehensive CoAP solutions and served as reference during the European Telecommunication Standards Institute (ETSI) *IoT Plugtests*, which represent the main interoperability testing event for the industry [61–64].

With the gained experience and the survey of HTTP server architectures, we were able to design a profound system architecture for CoAP-based IoT cloud services that focuses on the scalability issues when connecting a myriad of IoT devices. In turn, we re-designed Californium from scratch in 2013 with a new architecture. This Californium architecture achieves results that outperform all other available CoAP implementations as well as high-performance HTTP Web servers.

4.2.1 Design Goals

The main motivation for Californium is to provide a framework that allows for quick and easy implementation of CoAP servers, clients, and proxies. Furthermore, we aimed for the following characteristics while designing our software:

Usability

Usability for the IoT is the overall incentive for this thesis, and hence it also has a high priority for Californium. All protocol-specific mechanisms should be hidden underneath an intuitive API and be handled automatically as far as possible. For instance, the fragmentation of requests and responses into blockwise transfers can be fully transparent to the developer. Also re-registrations of interrupted observe relationships can be handled in the background. Developers should only need to implement handler functions for their Web resources and be confronted with abstracted request and response objects. Nonetheless, power-users must be able to tweak the configuration for their deployments programmatically or through a properties file, which can be created automatically to provide a list of all possible options.

Completeness

Functional completeness refers to the "degree to which the set of functions covers all the specified tasks and user objectives" [94]. For a CoAP framework, this means to implement all the features defined for the protocol and to support all high-level workflows:

1. Construct a server with customized resource handlers that is able to receive requests and send responses.
2. Let an application issue requests and wait for responses, either synchronous or asynchronous.
3. Combine these two workflows to construct intermediaries and hybrid client/server endpoints.

Sometimes requirements differ from the workflow intended by the protocol specification. Thus, there is a need for an advanced API that allows detailed access to the framework and protocol internals.

Maintainability

Completeness is directly connected to maintainability, as CoAP is a modular protocol that is continuously extended with new features for IoT and M2M applications. Chapter 2 illustrates the extent of this modularity and the available extensions. Reflecting this modularity in the system architecture allows Californium to be easier to maintain and enables better extensibility. For this, the multi-layer stack for CoAP from our initial implementation has proven well, since every extension corresponds to a single layer implementation. Other strategies come done to good software engineering such as the use of design patterns.

Scalability

Since Californium is primarily designed for unconstrained environments, it is supposed to address the scalability challenge for IoT cloud services. Although listed last, scalability is a central design goal and influenced the system architecture the most. In the following, we present our design for scalable IoT cloud services in detail.

4.2.2 System Architecture

Our system architecture for CoAP-based IoT cloud services is inspired by previous work for highly concurrent Internet services, in particular SEDA [192] and the PIPELINED architecture [37]. SEDA splits the message-handling process into multiple stages that are separated by event queues. Therefore, each stage can self-tune itself to have an optimal number of threads in its pool. The number of stages must be limited, though, to avoid too many context switches and bad cache locality. PIPELINED can be considered a special form of SEDA, as a pipeline is a chain of single-threaded stages. Belonging to the same pool, the threads of a pipeline have better cache behavior and best scalability is achieved with one pipeline per core. In addition, this architecture uses helper threads to execute blocking operations.

We propose a 3-stage architecture for CoAP-based IoT systems as depicted in Figure 4.7. It is mainly based on the lessons learned from our initial implementation that served as running code throughout the design phase of CoAP. Like SEDA, each stage is decoupled by queues and has its own thread pool. Its size does not depend on a dynamic scheduling policy, but on the static application requirements (e.g., complexity of specific resource handlers) and the execution platform (e.g., number of cores, CPU architecture, and operating system). This reduces complexity and the overhead of monitoring tasks. By default, the number of threads equals the number of cores and multiple messages can traverse our processing chain in parallel, similar to PIPELINED. For the business logic stage, we also allow for customized concurrency models, that is, developers can define multiple thread pools to wrap resources with blocking I/O calls or intensive calculations. In detail, the three stages function as follows:

4.2.3 Network Stage

The network stage (see Figure 4.7 bottom) is responsible for receiving and sending byte arrays over the network. It therefore abstracts the transport protocol, which is typically UDP or DTLS for CoAP. Micro-benchmarks show that using more than one thread to move data through the socket can increase the throughput on some platforms, but also decrease it on others. On Windows, for instance, using four receiver and four sender threads (4/4) instead of one each (1/1) almost doubles the achievable data rate of the provided UDP socket. On a Linux platform (RHEL6), however, increasing to two threads each (2/2) causes a 40% setback in throughput. Since we wrap the network I/O in its own independent stage, the server can chose an optimal number of threads for a specific platform without affecting other stages. By default, Californium uses one sender and one receiver thread per core on Windows and a single one each on Linux.

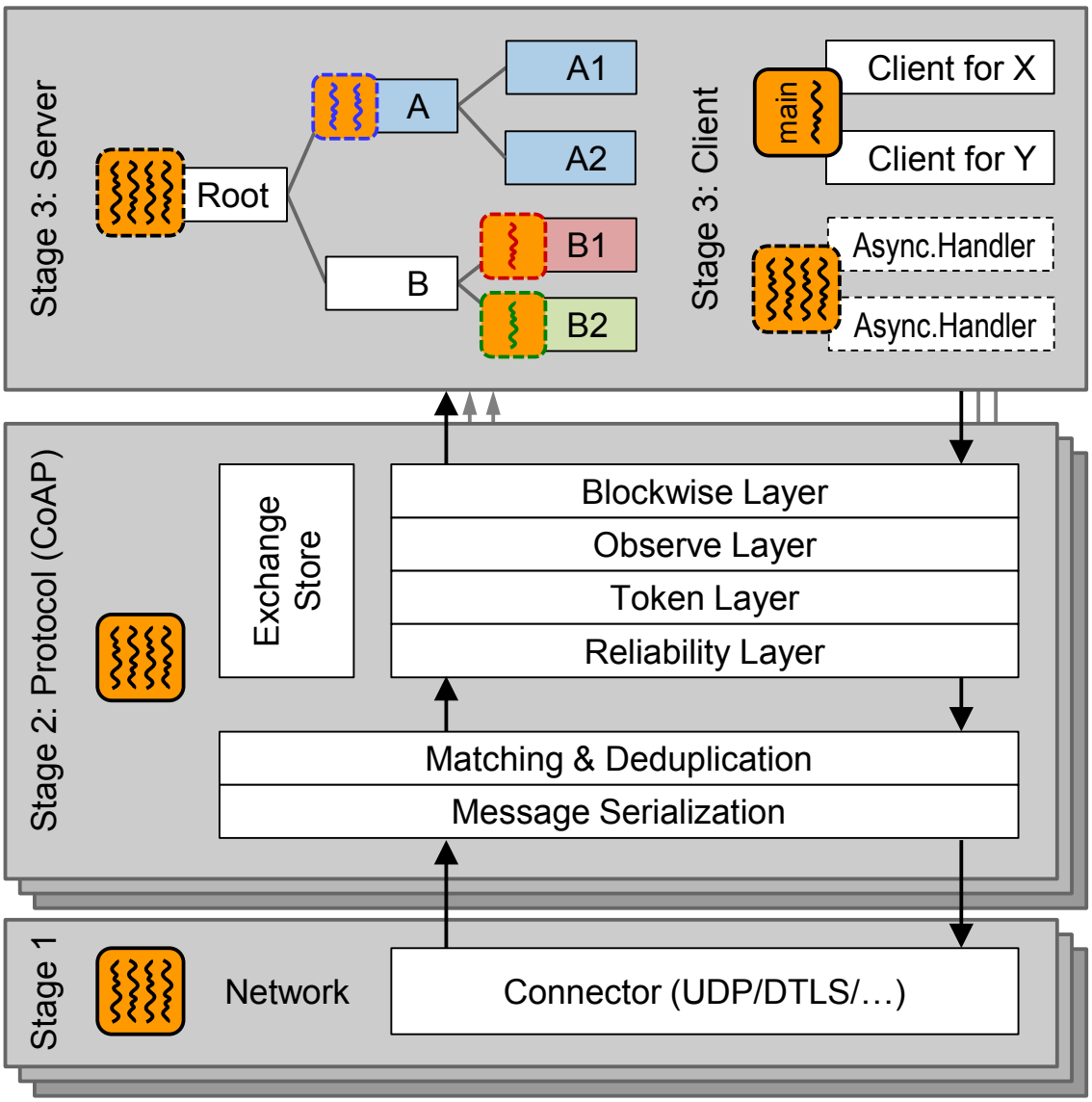


Figure 4.7: Our architecture has three stages with independent thread pools. The CoAP protocol is executed in the second stage: The Token Layer manages open request and connects them to the application; the Observe Layer handles observe relationships; if necessary, the Blockwise Layer deals with blockwise transfers (in atomic fashion [25]); the Reliability Layer manages retransmission timeouts. Message encoding and matching incoming messages to their state in the Exchange store happens outside this stack, so that it becomes a pure processing pipeline without synchronization overhead.

Yet the staged architecture provides the flexibility to implement more complex transports as well. Scandium (Sc) for instance, our DTLS-1.2 implementation, is implemented as extended network stage that uses additional worker threads to perform the security handshakes, encryption, and signing.

4.2.4 Protocol Stage

The protocol stage executes the CoAP protocol and has a thread pool with as many threads as cores (for all platforms). Internally, we use a multi-layer stack (see Figure 4.7) for good maintainability, where each layer implements a specific feature. In a previous version, each layer managed its specific state, e.g., the reliability layer held all timers for retransmissions and the blockwise layer tracked the current block number and the partially assembled payloads. This led, however, to several look-ups in the hashmaps of each layer when a message traverses the stack, which notably increases the processing time. Furthermore, distributing the state over several layers, which may include custom extensions, increases the risk of memory leaks due to missing clean-up mechanisms. Thus, for performance and correctness, it is crucial to separate bookkeeping from processing: our stack is a pure processing pipeline and all data and timers necessary for a request-response exchange are bundled in one corresponding object. When a message arrives, a matching step outside the stack accesses the exchange store only once to retrieve the necessary state or to create a new exchange object. The associated exchange is then passed along with the current message, layer by layer. And once an exchange completes, all connected state can easily be cleared from the central exchange store. For the deduplication, the state of incoming requests is kept even if the exchange is completed locally. This way, an endpoint can reply with the original response without re-executing the request. The state is eventually removed when the last message exceeds its lifetime.

4.2.5 Business Logic Stage

The business logic stage depends on the role of the system:

Servers host Web resources that implement handler functions to process the request and produces the respective response. Instead of a flat list of path strings, they are structured in a logical tree. This fits CoAP's strategy to encode individual path segments instead of the full path string. The tree structure makes it also straight-forward to implement dynamic resource-oriented architectures (ROAs). To store temporary data in resource state, for instance, handlers can dynamically add new sub-resources and delete them again.

However, resources can be configured to accept all requests for all its sub-resources. This is useful when hierarchical resource state is too large for an actual object tree, for example, entries in a resource directory (RD). The data would be stored in a database and the handler of the RD look-up resource would query the corresponding entries addressed through the URI.

By default, the server stage has no thread pool and the thread already used in the protocol stage invokes the resource handler. This saves costly context switches, in particular when the resource handler is simple. Developers can, however, choose individual concurrency models at each resource. To prioritize or balance Web resources, they can configure thread pools of different sizes or enforce a single-threaded environment in case a handler implementation is not thread-safe. If a resource does not define a thread pool, the thread pool of its parent, transitive ancestor, or eventually the protocol stage will be used. Figure 4.7, for instance, indicates a thread pool of size four that handles the root as well as resource B. Resource A defines its own thread pool of size two to execute its own handlers and those of its sub-resources. B1 and B2 enforce single-threaded execution of their handler.

Clients usually issue requests from the main thread or an explicit user thread. For this, we provide the `CoapClient` object API, which supports synchronous and asynchronous requests. Synchronous calls hand the request over to the protocol stage and block until the response is delivered by a protocol-stage thread. Asynchronous API calls return immediately and by default the protocol-stage thread will execute the response handler, which must be registered for asynchronous exchanges. Moreover, developers can also define different thread pools for each `CoapClient` object or define the same for multiple clients, for instance, to handle all incoming observe notifications by one pool to exploit cache-locality. This means that our architecture has the same processing behavior for response handling on the client side as for request handling on the server side.

The division into clients and servers is often relaxed in the IoT and endpoints usually have both roles [174]. To register a CoAP server at an RD, for instance, a POST request must be sent from the endpoint address (usually IP address and port) where the server is bound. This can be done in the main thread after the stages and their thread pools were initialized. When a request needs to be issued within a resource handler, it is possible to acquire a `CoapClient` object that is associated with the concurrency model of the Web resource, that is, the client uses the same thread pool and obeys the balanced resource quota defined by the developer.

4.2.6 Endpoints

A business logic might provide or rely on multiple CoAP endpoints, that is, to provide or reach CoAP Web resources via multiple ports (e.g., the default port and a 6LoWPAN compressible port in the range 61616–61631) and multiple transports such as UDP, DTLS, and SMS. Thus, we encapsulate the network stages together with their associated protocol stages in endpoint objects. The business logic stage can then connect to several endpoint objects in parallel. This way, alternative transports (UDP, DTLS, SMS, etc.) can have their individual variations of the CoAP stack (e.g., no message sub-layer for the reliable SMS and TCP transports). Moreover, CoAP hosts can easily distinguish between different network interfaces to send a reply through the correct socket, so that the recipient is able to match the message. This addresses a problem introduced by many OS APIs where the datagram does not provide the information to which socket address it was sent.

4.3 Californium (Cf) Implementation

With the *Californium (Cf) CoAP framework*, we provide a reference implementation of our architecture. We decided to use the Java language because of its portability, its broad developer base, and its language-support for parallel processing and hence good performance on multi-core systems. The primary target of Californium are server platforms in the cloud. With optimized JVMs such as in the *Oracle ARM JDK* or the *PreonVM*[21], however, our framework also runs on Class 2 IoT devices.

Due to its popularity for IoT projects, Californium was introduced to the Eclipse Foundation in 2014 to foster further development.[22] The source code is publicly available on *GitHub* under EPL[23]+EDL[24] dual-licensing.[25] Usually, other projects include our framework using its *Maven*[26] artifacts that are available on Maven Central[27].

[21] http://www.virtenio.com/en/products/virtual-machine.html (accessed on 12 Feb 2015)
[22] https://www.eclipse.org/californium/ (accessed on 12 Feb 2015)
[23] http://www.eclipse.org/org/documents/epl-v10.php (accessed on 12 Feb 2015)
[24] http://www.eclipse.org/org/documents/edl-v10.php (accessed on 12 Feb 2015)
[25] https://github.com/eclipse?query=californium (accessed on 12 Feb 2015)
[26] http://maven.apache.org/ (accessed on 12 Feb 2015)
[27] http://search.maven.org/ (accessed on 12 Feb 2015)

4.3.1 Classes Overview

Figure 4.8 shows how our architecture can be realized in more detail. Our `CoapEndpoint` class heavily uses the strategy design pattern, for instance, to select the right network stage. This is implemented in the `Connector` class, which is depicted at the bottom of the figure. By default, this is the `UDPConnector` from our *element-connector* Maven artifact. It can easily be exchanged with the `DtlsConnector` from our *Scandium (Sc)* project, which provides an implementation of DTLS 1.2. The raw data coming in through the `Connector` enters the `InboxImpl` (an implementation of the `RawDataChannel` Java interface of the element-connector), where it is enqueued for parsing to a CoAP message. The endpoint class uses a `ScheduledExecutorService` to implement queue and thread pool for the protocol stage. Using a protocol thread, the `InboxImpl` checks the list of registered `MessageInterceptors`, which can be used to log the ongoing traffic, filter specific messages, or even manipulate them. When the message was not cancelled, it is handed over to the `Matcher` to retrieve the existing exchange state. If there is none, a new `Exchange` object is created. The `Matcher` also holds the implementation of the deduplication strategy. By default, Californium uses a mark-and-sweep algorithm to remove all expired exchange state according to the message lifetime defined in the CoAP specification. The state of exchanges that originate locally is removed on completion or an application-specific timeout to handle lost separate responses.

When the incoming message is paired with its exchange state, both are handed to the `CoapStack`. This class reflects the multi-layer design to implement the protocol. Depending on the `Connector`, the `CoapEndpoint` selects the corresponding layers. For the default UDP and DTLS connectors, these are the same as described in Figure 4.7 in the architecture section. A multicast connector for CoAP group communication would, for instance, require a layer to implement response suppression [158].

At the top of the stack sits the `MessageDeliverer`. When the message is a request, it resolves the Web resource for the addressed URI and delivers the request. If the resource has its own concurrency model, the corresponding `Executor` is used the call the handler method. Otherwise, it is directly called from the protocol thread. When the message is a response, the `MessageDeliverer` updates the status of the respective request. If the request was made through a synchronous call, this will unblock the original thread. If it was an asynchronous call, the `CoapHandler` is called in the same manner as a Web resource (i.e., using the configured concurrency model).

On the way down the stack, the `StackTopAdapter` makes sure that locally created requests are associated with an exchange. The reason for this is that from the client API perspective, there is no exchange state, just a request and in the case of success a

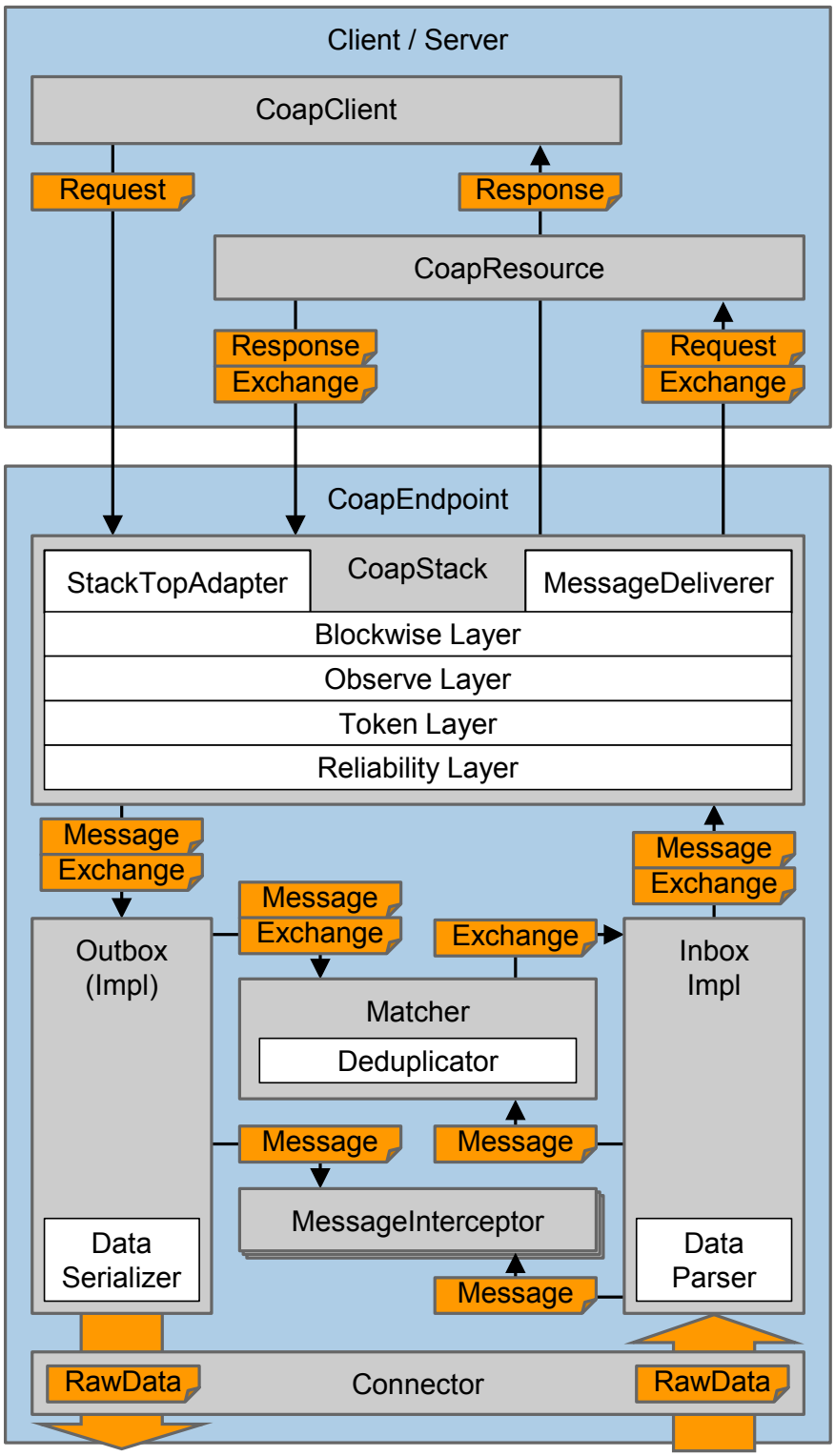


Figure 4.8: CoapEndpoint

corresponding response. The exchange state is only required for the framework internals. The `OutboxImpl` is the counter-part to the `InboxImpl`. It stores the exchange state of outgoing messages through the `Matcher`. This bookkeeping is required to match incoming responses, to handle blockwise transfers, and to allow the deduplication mechanism to send out the original reply when a duplicate arrives (e.g., to send the same piggybacked response in case the ACK to the CON request was lost). The outbox also processes the `MessageInterceptors` for outgoing messages before serializing them and passing `RawData` objects with the byte array and addresses to the queue of the `Connector`.

4.3.2 API Examples

For good usability, we aim for an intuitive API that aligns with well-known patterns from the Web. For servers, developers need to implement resource handler methods, similar to other Web server frameworks. This is done in application-specific resource classes that extend `CoapResource`. For custom concurrency models, we provide the `ConcurrentCoapResource` class for extension. Developers then only need to add their Web resources to a `CoapServer` instance and start it:

```
1  import org.eclipse.californium.core.CoapResource;
2  import org.eclipse.californium.core.CoapServer;
3  import org.eclipse.californium.core.server.resources.CoapExchange;
4
5  import static org.eclipse.californium.core.coap.CoAP.ResponseCode.*; // shortcuts
6
7  public class MyResource extends CoapResource {
8
9    @Override
10   public void handleGET(CoapExchange exchange) {
11     exchange.respond("hello world"); // reply with 2.05 payload (text/plain)
12   }
13
14   @Override
15   public void handlePOST(CoapExchange exchange) {
16     exchange.accept(); // make it a separate response
17
18     if (exchange.getRequestOptions()....) {
19       // do something specific to the request options
20     }
21     exchange.respond(CREATED); // reply with 2.01 response code only
22   }
23
24   public static void main(String[] args) {
25     CoapServer server = new CoapServer();
26     server.add(new MyResource("hello"));
27     server.start(); // does all the magic
28   }
29 }
```

Listing 4: Californium server API

For clients, we aimed for a fluent API, where developers only need to know a single class to build their requests and receive the responses. The auto-complete feature of modern integrated development environments (IDEs) will show all possible calls provided by the `CoapClient` object API. Arguments and return values directly point the developer to the related API classes, `CoapResponse`, `CoapHandler`, and `CoapObserveRelation`. The `CoapClient` constructor takes the URI of the target resource, but it can also be changed later on. The client then interact using the RESTful verbs on the Web resource:

```
1  public static void main(String[] args) {
2    CoapClient client1 = new CoapClient("coap://iot.eclipse.org/multi-format");
3
4    String text = client1.get().getResponseText(); // blocking call
5    String xml = client1.get(APPLICATION_XML).getResponseText();
6
7    CoapClient client2 = new CoapClient("coap://iot.eclipse.org:5683/test");
8
9    CoapResponse resp = client2.put("payload", TEXT_PLAIN); // for response details
10   System.out.println( resp.isSuccess() );
11   System.out.println( resp.getOptions() );
12
13   client2.setURI("coap://iot.eclipse.org/validate");
14   client2.useNONs();
15   client2.delete();
16
17   client1.setURI("coap://iot.eclipse.org/large");
18   client1.useCONs().useEarlyNegotiation(32).get(); // it is a fluent API
19 }
```

Listing 5: Synchronous client API

The synchronous call above will block until the response arrives. For asynchronous calls, the developer must provide a `CoapHandler` for the response. By default, it is executed by a thread from the protocol stage:

```
1  public static void main(String[] args) {
2    CoapClient client = new CoapClient("coap://iot.eclipse.org/separate");
3
4    client.get(new CoapHandler() { // e.g., anonymous inner class
5      @Override
6      public void onLoad(CoapResponse response) { // also error resp.
7        System.out.println( response.getResponseText() );
8      }
9
10     @Override
11     public void onError() { // I/O errors and timeouts
12       System.err.println("Failed");
13     }
14   });
15   // ...
16 }
```

Listing 6: Asynchronous client API

Observing works similar to asynchronous calls, only that the `observe()` method will return a `CoapObserveRelation` handle that can be used to cancel the registration:

```
public static void main(String[] args) {
  CoapClient client = new CoapClient("coap://iot.eclipse.org:5683/obs");

  CoapObserveRelation relation = client.observe(new CoapHandler() {
    @Override
    public void onLoad(CoapResponse response) {
      System.out.println( response.getResponseText() );
    }

    @Override
    public void onError() {
      System.err.println("Failed");
    }
  });
  // ...
  relation.proactiveCancel();
}
```

Listing 7: Californium observe API

4.3.3 Lessons Learned

The implementation of CoAP throughout its standardization process has been helping us to better understand the trade-offs and challenges. One of the main take-away points is that a full-featured protocol implementation is not as straight-forward as one would assume. Although the basic protocol mechanisms can be implemented within a couple of hours, more and more complications appear when adding the reliability mechanisms (e.g., deduplication and message lifetimes) and extensions.

State Management

The observe and group communication extensions result in multiple responses, and blockwise transfers divide the logical request and response into multiple messages. For this reason, we use an exchange object that holds the state and covers all associated messages that are passed between client and server. This also has led to the separation of bookkeeping and processing. It is more efficient having a central look-up step and passing around the exchange object than organizing the relevant information per feature, such as having a separate block management component. The observer design pattern has turned out well to propagate the many critical changes that can happen throughout the processing pipeline. We use message observers to notify the API of progress and exchange observers to properly clean up the exchange store.

Race Conditions

A challenge for CoAP that only arises in unconstrained environments are node-local race conditions for the protocol. As UDP datagrams can get reordered, observe notifications, for instance, carry a strictly increasing sequence number in the `Observe` option. Only the latest notification with a higher sequence number may be accepted and older ones must be dropped to allow for eventual consistency, that is, eventually having the latest representation of a resource. In multi-threaded environments, however, this might be violated due to race conditions: When a resource wants to send two notifications within a short time, it can happen that two different protocol-stage threads process one notification each. Without further information, it is not possible for the two threads to know which notification was produced first by the resource. Even if they use an atomic counter and assign different observe-numbers to the notification they process, they might have mixed up the order. Therefore, it is essential that the resource itself sets the observe option on creation—and has a thread-safe implementation itself.

On the client side, multiple notifications might arrive at the same time and can be processed by two different threads. At some point, each thread has to decide whether its notification is new or obsolete. The first thread will assert its notification to be new, however, shortly afterwards, the second thread determines that its notification is even newer. Yet due to the non-deterministic scheduling-behavior, the second thread might execute the notification handler of the application first. Nonetheless, the first thread already decided that its notification is new and will overwrite the resource state of the client with its older representation when executing the notification handler after the second thread. If no new notification follows for a while, the client will work with an incorrect state. Thus, the notification handler of the application needs to be a critical section and must do the reordering.

Deduplication State

The main bottleneck of our system turns out to be the memory required for deduplication. When a duplicate CON request arrives to which the server already sent a response, we must retransmit it without re-executing the request (at-most-once semantics). Thus, along with the request information for filtering, CoAP servers have to cache the response for retransmission. These data have to be stored in the exchange store for the maximum $EXCHANGE\_LIFETIME = 247\ s$ [172]. On an 64-bit JVM, each field occupies at least 8 bytes, even when it is a null reference. Therefore, an Exchange object for a request with a simple destination URI and a response with even no payload occupies around half a kilobyte. When our system processes 400,000 requests per second, it

would require 64 GiB for the 100 million exchange store entries that accumulate. Thus, the exchange state must be reduced and compressed after finishing the request and be dropped as soon as possible. NON requests are less critical, as they have maybe semantics and only need duplicate filtering: No response needs to be cached and their lifetime is only 145 seconds. CoAP servers can further relax deduplication. When modelling the application with idempotent requests only, Californium can be configured without deduplication (using the `NoDeduplicator` class). In a RESTful design, GET, PUT, and DELETE must always fulfill idempotence. Then no responses have to be cached and computation intensive resources can cache their results locally. POST can be optimized in a similar way, where the resource handler exploits the knowledge about the application to implement a more efficient deduplication strategy. These are techniques that are already discussed in Chapter 3. Strategies that enable Web technology for resource-constrained devices also apply for comparably unconstrained environments, where the myriad of connected devices push systems to their limits as well.

4.3.4 Provided Tools

Our Californium framework already provides a set of test programs and tools. Next to the core libraries and examples, the main repository[28] also contains implementations of the ETSI Plugtest specification, which are used for interoperability testing [61–64]. The `cf-plugtest-client` provides the required client functionality and `cf-plugtest-server` the corresponding Web resources. `cf-plugtest-checker` automatically tests a server implementation whether it handles the specified test requests correctly.

The tools repository[29] offers basic tools to work with CoAP. It contains a command line interface (CLI) client comparable with *cURL*[30] (`cf-client`), a basic graphical browser client (`cf-browser`), and a stand-alone resource directory (RD) (`cf-rd`). Furthermore, it provides the *CoAPBench* benchmarking tool (`cf-coapbench`), which we use in the following evaluation.

[28] https://github.com/eclipse/californium (accessed on 12 Feb 2015)
[29] https://github.com/eclipse/californium.tools (accessed on 12 Feb 2015)
[30] http://curl.haxx.se/ (accessed on 12 Feb 2015)

4.4 Evaluation and Results

We use our Californium reference implementation to evaluate our system architecture for scalable IoT cloud services. For this, we perform benchmarks and compare the results to the state of the art, both high-performance HTTP Web servers and other CoAP solutions for unconstrained environments. Due to the different communication models that can be found in IoT applications, there are multiple possible evaluation scenarios:

1. *Cloud service as server*: This is the typical scenario for the Web 2.0 and HTTP-based IoT applications. Since HTTP is missing an efficient server push mechanism, IoT devices are usually programmed as HTTP clients that POST their data to the service whenever they have an update. In between, devices can go into sleep mode to conserve energy. For CoAP, this scenario applies for resource directories (RDs), which are servers that are used for device management and discovery [170]. IoT devices register there on start-up by sending a request and periodically update their status. In addition, other devices and services contact the RD server to perform look-ups.

2. *Cloud service as client*: With CoAP's push mechanism, the server role has become practical for IoT devices. The cloud service takes over the role of a Web mashup engine that is client to many servers that run on the resource-constrained devices. It sends requests for configuration and actuation, and observes resources for monitoring and sensing tasks. Without server role, services need to make use of multicast discovery or contact an RD that runs as separate service.

3. *Cloud service as both*: Complex business logic requires both roles by the service, e.g., to observe device resources and to provide computed results again as resources for other services. For instance, the OMA OMA Lightweight M2M (LWM2M) specification is built around this scenario. The LWM2M server is a resource directory and proxy that receives registration and look-up requests from devices, but also issues requests to the resources of the devices. This means that IoT devices are hybrids as well: they are primarily CoAP servers to provide their data, but use requests to register with the service (and thereby open ports in firewalls).

In our evaluation, we choose the *cloud-service-as-server* scenario because it allows for a direct comparison with HTTP servers, the current state of the art in scalable service design.

4.4.1 Experiment Setup

While there are plenty of benchmark tools available for HTTP, to the best of our knowledge, there is none for CoAP. Therefore, we developed *CoAPBench*, a tool similar to ApacheBench[31]. It is part of our Californium framework, and hence publicly available to replicate our experiments.

CoAPBench

CoAPBench uses virtual clients to meet the defined concurrency factor. To have enough resources to saturate the server and keep all collected statistics in memory, CoAPBench can be distributed over multiple machines. A master controls the benchmark by establishing a TCP connection to all slave instances. We designed this master/slave mechanism to be able to execute third-party benchmark tools as well. Thus, we can run ApacheBench distributed and synchronized over multiple machines and bring even very powerful HTTP servers into saturation. Note that master and slaves only communicate before and after the experiment, so that the network traffic is not influenced by our tool.

CoAPBench adheres to basic congestion control, that is, each CoAP client sends Confirmable requests and waits for the response before the next request is issued. We disable retransmissions, though, to not blur the numbers of sent and successfully handled requests. In case of message loss, a client times out after 10 seconds, records the loss in a separate counter, and continues with a new request.

Setup

All benchmarks are performed using the CoAPBench tool, which runs distributed over three machines. For HTTP, it executes ApacheBench in distributed fashion to be able to fully saturate the servers. Figure 4.9 depicts the setup in more detail and gives the machine specifications. The platform hosting the Web server under test varies and is given for the individual experiments.

The evaluation focuses on the performance and scalability of the protocol handling by the systems—not the business logic. Thus, CoAPBench issues simple GET requests to a `/benchmark` resource, which responds with a short "hello world". CoAP and HTTP requests and responses are semantically equal, that is, they hold the same payload and metadata such as header fields or options.

[31] http://httpd.apache.org/docs/2.4/programs/ab.html

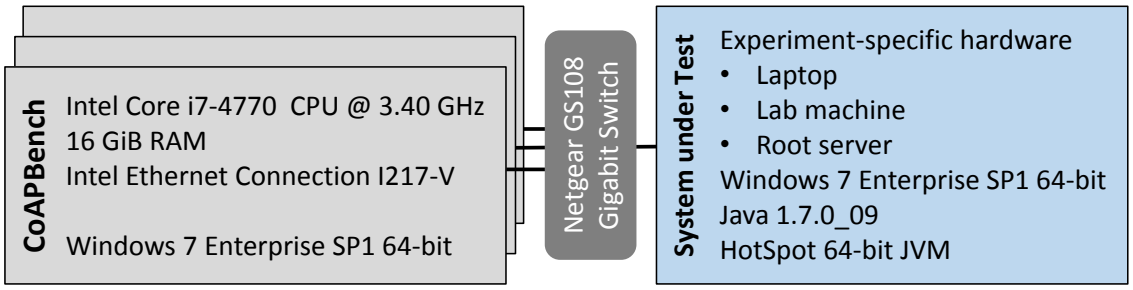


Figure 4.9: All CoAP and HTTP servers are tested in this general setup. The hardware for the system under test depends on the individual experiment and is given in the corresponding sections.

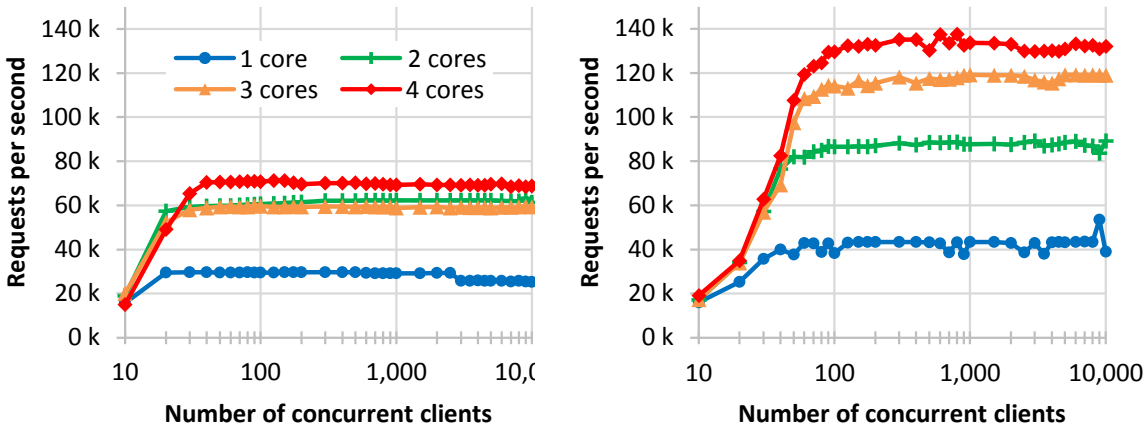
The benchmark stresses the server for 60 seconds with requests followed by a 15 seconds cool-down period. To evaluate the service scalability, we use a growing number of simultaneous clients, whereas the concurrency factor is increased stepwise from 10 to 10,000. To have deterministic results, we disable Hyper-Threading and Turbo-Boost on the machine hosting the system under test. In particular the Turbo-Boost technology causes highly non-deterministic results, as the clock speed depends on the load and temperature of individual cores.

4.4.2 Scalability Verification

First, we evaluate whether the Californium architecture meets our design goal of scalability. It is supposed to scale well with an increasing number of available CPU cores. Furthermore, it must exhibit a stable throughput at high concurrency factors. For Californium, we use a $r/s/p$ notation where r is the number of receiver threads, s the number of sender threads, and p the number of protocol threads. By default they equal the number of available cores.

Multi-core Utilization

Our design specifically focuses on the utilization of modern multi-core systems. We evaluate this by measuring the throughput with different processor affinity settings, which is directly provided through the Task Manager on Windows platforms. We compare the results of our *Californium* architecture to the initial framework design, hereinafter referred to as *Initial-Cf*. For these benchmarks, the two CoAP servers are running on a quad-core laptop machine with an Intel i7-3720M processor at 2.6 GHz, 24 GiB of RAM, and Intel 82579LM Gigabit Ethernet adapter.



(a) Initial-Cf

(b) Californium

Figure 4.10: We evaluate the throughput for different numbers of assigned CPU cores of a quad-core laptop machine. Both CoAP servers always stay stable over an increasing concurrency factor. The Californium system architecture achieves a much higher throughput, though.

Figure 4.10 shows that our Californium architecture scales well with the number of available cores. Separating bookkeeping from processing allows multiple threads to work independently with little synchronization overhead. Using two instead of a single core almost exactly doubles the throughput. On four cores, we perform about 3.4 times better than on a single core, which is reasonable since not all tasks can be parallelized. Socket I/O, for instance is partly done in the kernel and always runs on *Core 0* on Windows machines. Our maximum throughput (on this machine) with a $4/4/4$ configuration for four cores is 137,592 requests per second versus 71,255 requests per second for the Initial-Cf design. Primarily, this increase comes from the more efficient processing pipeline of our architecture. Yet we were also able to improve socket I/O, which is often the main bottleneck in implementations.

Platform Independence

To ensure our architecture is independent from the underlying hardware, we conduct experiments with different platforms for the system under test. For this, we compare Californium to our *Initial-Cf* and two other available CoAP solutions that provide a similar scope of operation, *Sensinode NanoService Platform* [112] and *nCoAP*[32]. The general setup and benchmark configuration is as described in Section 4.4.1.

[32] https://github.com/okleine/nCoAP (accessed on 12 Feb 2015)

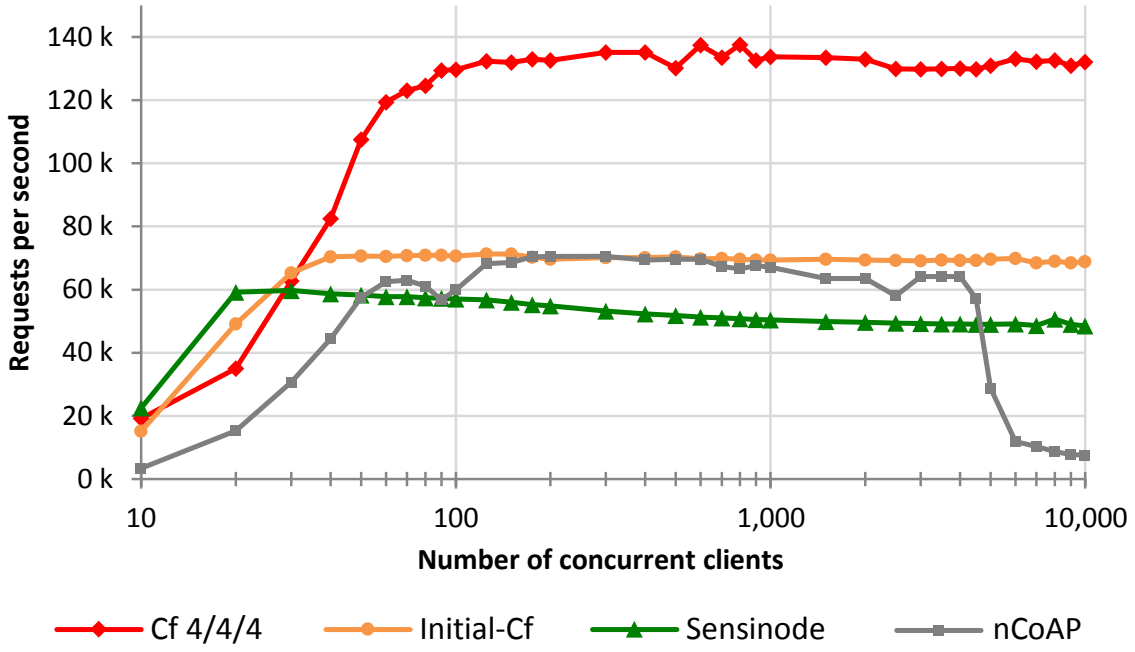


Figure 4.11: Throughput on a quad-core laptop with Intel Core i7-3720M at 2.6 GHz: Here, Californium in its default configuration (Cf 4/4/4) is about twice as fast as comparable CoAP solutions. A staged architecture cannot fully utilize the CPU under minimal loads, and hence the throughput grows slightly slower. At low concurrency factors, Californium still handles the requests in under one millisecond, though.

Figure 4.11 shows the results for the quad-core laptop machine from the previous experiment in comparison with the other CoAP implementations. On this platform, Californium is about twice as fast as the other solutions. Compared to Sensinode, we pay for the high throughput under heavy load with a slightly slower growth in the beginning. When there are only a few incoming requests, we are not able to fully utilize the CPU. This has two reasons: Sensinode has a strict SPED architecture that uses a single thread, which is bound to a single core. Californium uses all four cores, which however, causes context switches that need to transfer data to the caches of other cores. In addition, we have a message queue between network and protocol stage (but no queue toward the business logic stage, since there is no need to allocate a thread pool for the simple benchmark resource), which increases the total processing time of a request further. Since CoAPBench always waits for a request to finish before issuing the next one, the clients spend slightly more time waiting for the response. Still, 99% of all requests finish in under one millisecond for low concurrency factors.

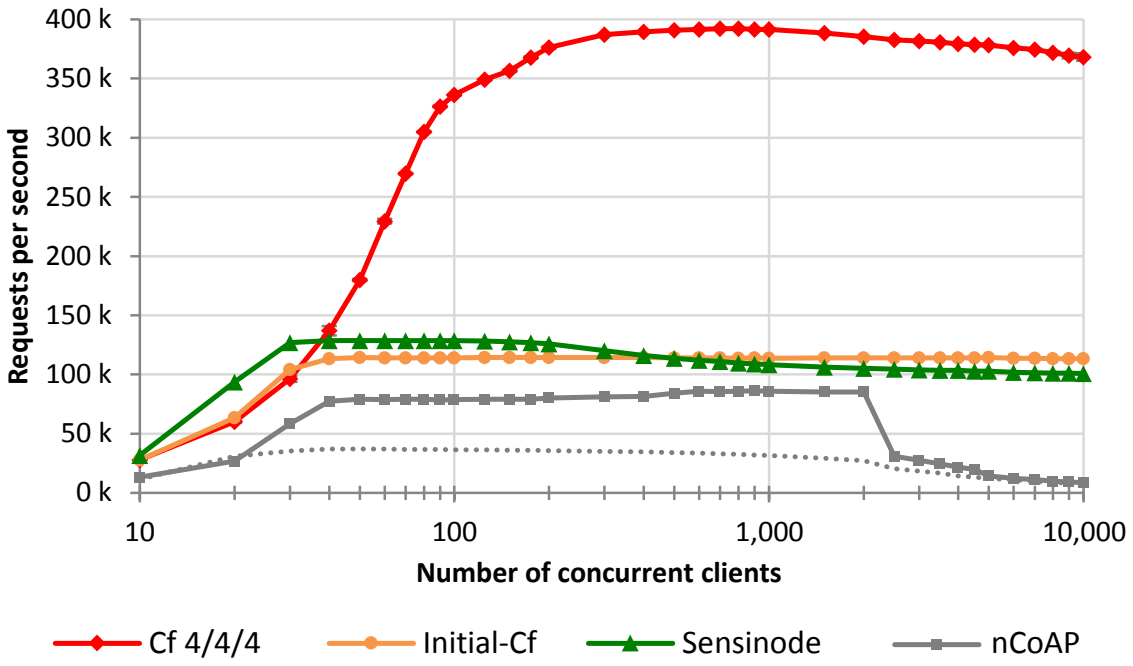


Figure 4.12: Throughput on a quad-core lab machine with Intel Core i7-4770 at 3.40 GHz: On powerful systems, Californium (Cf 4/4/4) is about three times as powerful as the other implementations. The performance drop of nCoAP is caused by inefficient state management. Once the system runs out of memory, the throughput drops for all concurrency factors as indicated by the dotted line.

Figure 4.12 summarizes the benchmarks on a quad-core lab workstation with Intel Core i7-4770 CPU at 3.40 GHz, 16 GiB of RAM, and Intel Ethernet Connection I217-V. This is the same system as used for the CoAPBench clients (see Figure 4.9). The general result resembles the trend of the previous experiment. Due to the higher clock speed, the throughput is much higher, though. Californium tops at 392,927 requests per second versus 131,327 for Sensinode on second place. Again, our system grows more slowly and needs about 100 concurrent clients to saturate the CPU.

Interestingly, nCoAP again significantly drops in performance at higher concurrency factors, this time at 2,000 simultaneous clients. Using ten consecutive runs in this experiment, we confirm that nCoAP suffers from this drop for all consecutive runs, even for low concurrency factors. The ten runs average is shown as the dotted gray line in Figure 4.12. On the laptop machine with 24 GiB, the performance drop was at 4,500 simultaneous clients (see Figure 4.11). This hints at a memory leak or problem in the memory management of nCoAP.

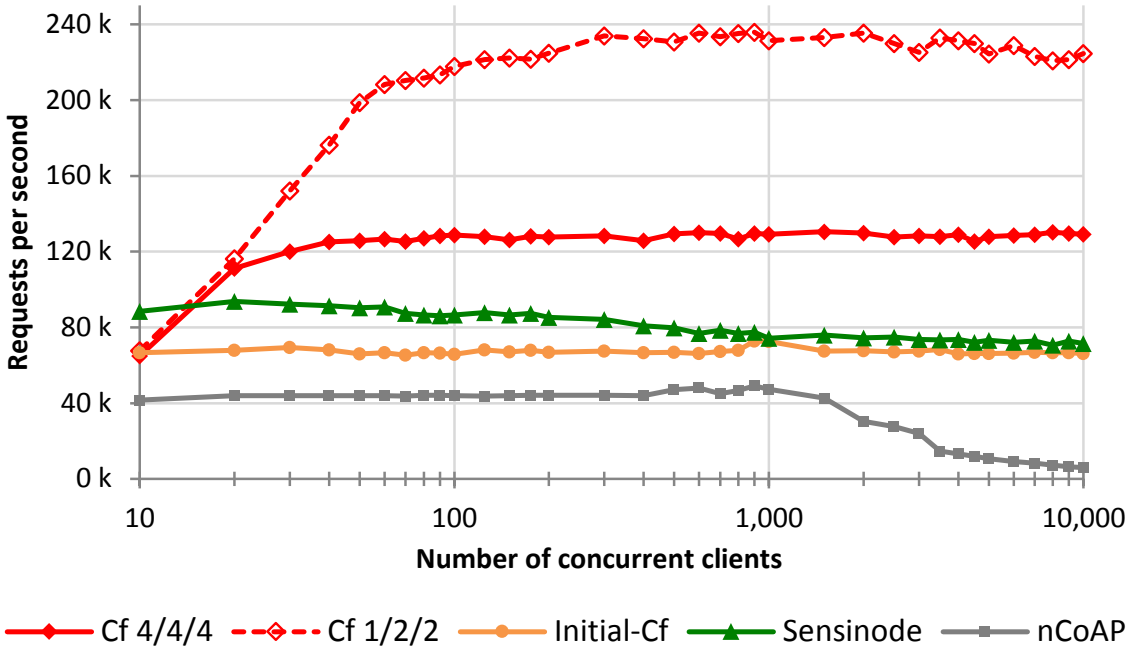


Figure 4.13: Throughput on a quad-core server with Intel Xeon E3-1265LV2 at 2.5 GHz: The high-performance bus for the network card interface reduces the I/O bottleneck and all systems start with higher throughput. The results also show that the performance primarily depends on the clock speed.

Finally, we evaluated the hardware independence of our architecture on a commercial quad-core root server, the Kontron SYMKLOUD with one Intel Xeon E3-1265LV2 CPU at 2.5 GHz, 8 GiB of RAM, and 10-gigabit Ethernet. Because of the fixed setup at a remote location, there was only a single physical machine available to host CoAPBench. It runs on an identical machine that is directly connected over 10-gigabit Ethernet. Furthermore, we were not able to disable Hyper-Threading and Turbo-Boost.

The performance of our system and the overall trend is confirmed for this platform as well (see Figure 4.13). The Cf 4/4/4 configuration, however, does not fully utilize the CPU, which can be caused by two factors: CoAPBench running on only one machine instead of three and the Xeon CPU architecture. Thus, conclusions from a direct comparison with the previous experiments must be treated with caution. Nonetheless, the experiment shows how our flexible concurrency model allows to adapt to different environments. Changing the configuration to one receiver thread, two sender threads, and two protocol threads (Cf 1/2/2) results in almost doubling the throughput compared to the default configuration. The other CoAP solutions do not provide such configuration options. This confirms the platform independence of our system architecture.

4.4.3 State-of-the-art Throughput

Next, we evaluate how CoAP-based IoT cloud services perform in comparison to state-of-the-art HTTP solutions. For this experiment, we compare our system to five popular HTTP servers: *Apache HTTP Server (2.4.6)*[33], *Tomcat (7.0.34)*[34], *Node.js (0.10.20)*[35], *Grizzly (2.3.6)*[36], *Jetty (9.1.5)*[37], and *Vert.x (1.3.1-final)*[38]. We also include the other three CoAP implementations (*Initial-Cf*, *Sensinode NanoService Platform*, and *nCoAP*) for reference. To have a controlled environment for the system under test, we use our lab machines with Intel Core i7-4770 CPU at 3.40 GHz, 16 GiB of RAM, and Intel Ethernet Connection I217-V. The setup is as depicted in Figure 4.9 and both Hyper-Threading and Turbo-Boost are disabled. For a fair comparison, we optimize each server configuration for this lab machine following the provided documentations. To achieve the best result for each system, we also request the 'natural' resource of the server: for Apache, this is `/benchmark/index.php`, for Tomcat and Node.js `/benchmark/` (with slash), and `/benchmark` for all other servers.

With Keep-alive

First we evaluate the performance with the keep-alive option of HTTP/1.1. Here, a client establishes a single TCP connection for all subsequent requests. This saves costly RTTs for the handshake and remedies the slow-start mechanism of TCP. As seen in Figure 4.14, Vert.x performs best among the HTTP solutions in this mode. It impressively solves the so-called *C10K problem*, that is, being able to maintain 10,000 concurrent TCP connections to its clients at high throughput. Vert.x is the only server with high standard deviation, though. We indicate this using error bars with $\pm 1\sigma$. For most other series, the standard deviation is negligible, so their error bars are omitted for a clearer figure. Jetty also shows stable performance for high concurrency factors, which is why it is popular for IoT applications as well. Tomcat has good performance on its first run, but automatically disables keep-alive once it experiences high concurrency factors. The throughput consequently drops for all subsequent runs, which we indicate by the dotted light blue line. Thus, we limit the number of concurrent clients to 200 to keep Tomcat in the range it is originally designed for. Its successor for Java Enterprise application servers, Grizzly, scales better and only gives in at around 5,000 simultaneous clients. The Apache Web server with PHP is not designed for highly scalable cloud services. However, Apache is still the most popular

[33] http://httpd.apache.org/ (accessed on 12 Feb 2015)
[34] http://tomcat.apache.org/ (accessed on 12 Feb 2015)
[35] https://nodejs.org/ (accessed on 12 Feb 2015)
[36] https://grizzly.java.net/ (accessed on 12 Feb 2015)
[37] http://www.eclipse.org/jetty/ (accessed on 12 Feb 2015)
[38] http://vertx.io/ (accessed on 12 Feb 2015)

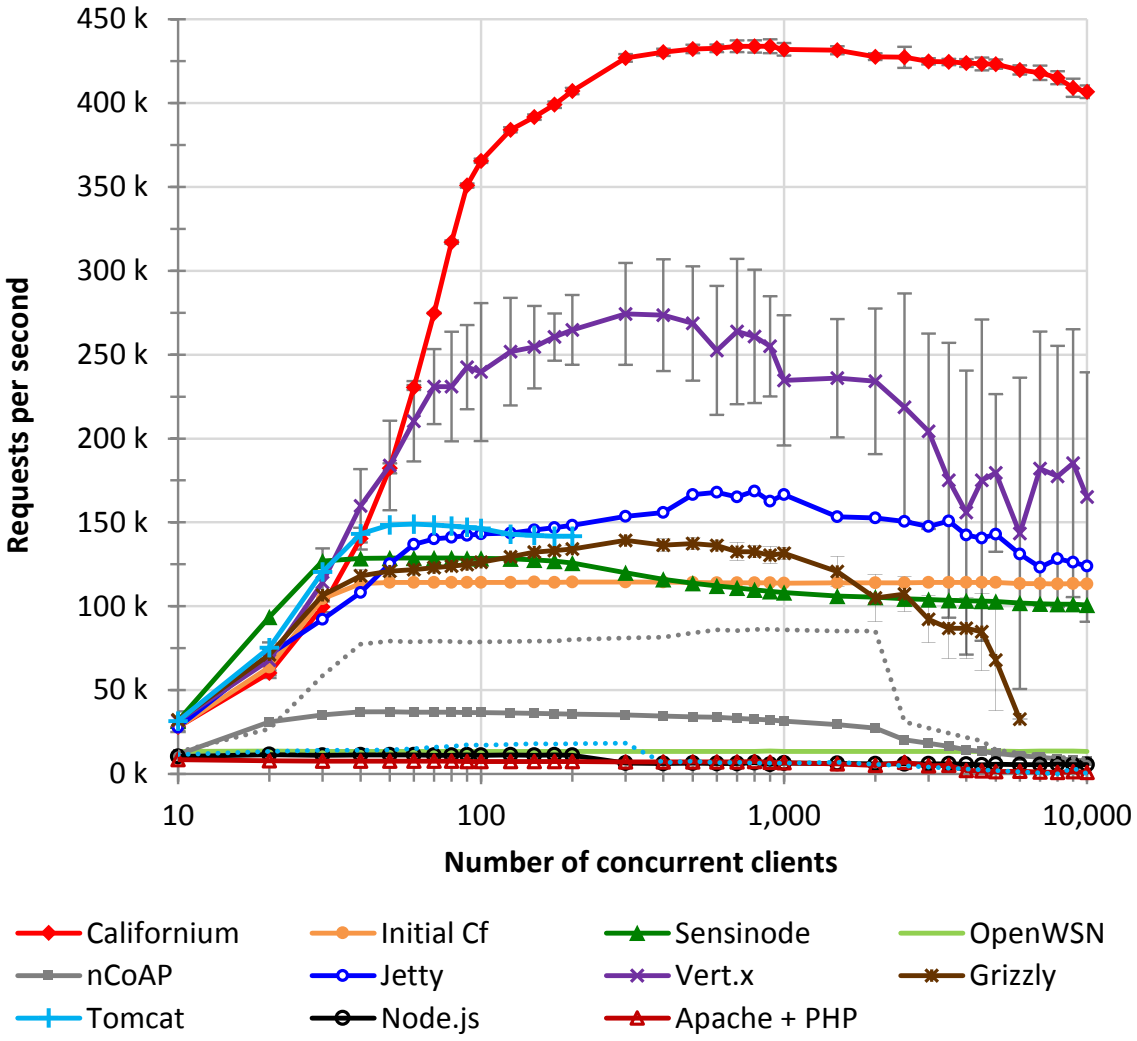


Figure 4.14: Throughput with HTTP keep-alive: Modern architectures for HTTP servers can handle a high number of parallel TCP connections. However, CoAP in general scales better for high concurrency factors and our system can handle the most requests per second.

Web server and is included for reference. It achieves a steady throughput at about 7,000 requests per second until 3,500 concurrent clients. Beyond that point, the performance slowly declines.

The Initial-Cf and Sensinode CoAP servers exhibit similar performance as Grizzly, but keep stable toward the end. Sensinode handles everything in a single thread, though, and could significantly benefit from service replication. Overall, our system performs best with up to about 400,000 requests per second and stable throughput for high concurrency factors. We also indicate the standard deviation for our system. It is so small, however, that the error bars are mostly hidden behind the data points.

Without Keep-alive

In an IoT scenario, devices often close the connection after each request to resume sleep. Furthermore, we expect way more IoT devices interacting with a single cloud service than open TCP connections can be handled by the operating system. For instance, public cloud services cannot maintain a TCP connection to every device they serve due to the sheer number. As a consequence, we focus on the throughput behavior without the keep-alive option, that is, a new TCP connection for each request. Moreover, high message rates in the IoT originate from tens of millions of devices sending alternately in minute or hour intervals, rather than tens of thousands sending constantly at very high rate. Scenarios for this are sensors deployed throughout a smart metropolis or smart metering for real-time demand side management.

Figure 4.15 shows that HTTP suffers from the overhead of TCP, whose avoidance was in fact one of the design goals behind CoAP. Here, we use a logarithmic scale also on the y-axis to cover all results in a single graph. Apache actually performs similar with or without keep-alive, so its series can be used for reference when comparing Figure 4.14 and Figure 4.15. The CoAP results are not affected by the change, since it is connectionless, and we can use the measurements reported above for comparison.

Having a stable throughput at high concurrency factors, Node.js now performs best among the HTTP servers. The cluster mode has good scalability for short-lived TCP connections and can handle almost 6,000 requests per second at 10,000 simultaneous clients. Node.js is followed by Tomcat, Apache, and Grizzly, which all converge toward about 3,500 requests per second at the end. Without keep-alive, the servers designed to overcome the C10K problem (Vert.x and Jetty) drop from 10,000 to about 2,000 requests per second already for more than 50 concurrent clients.

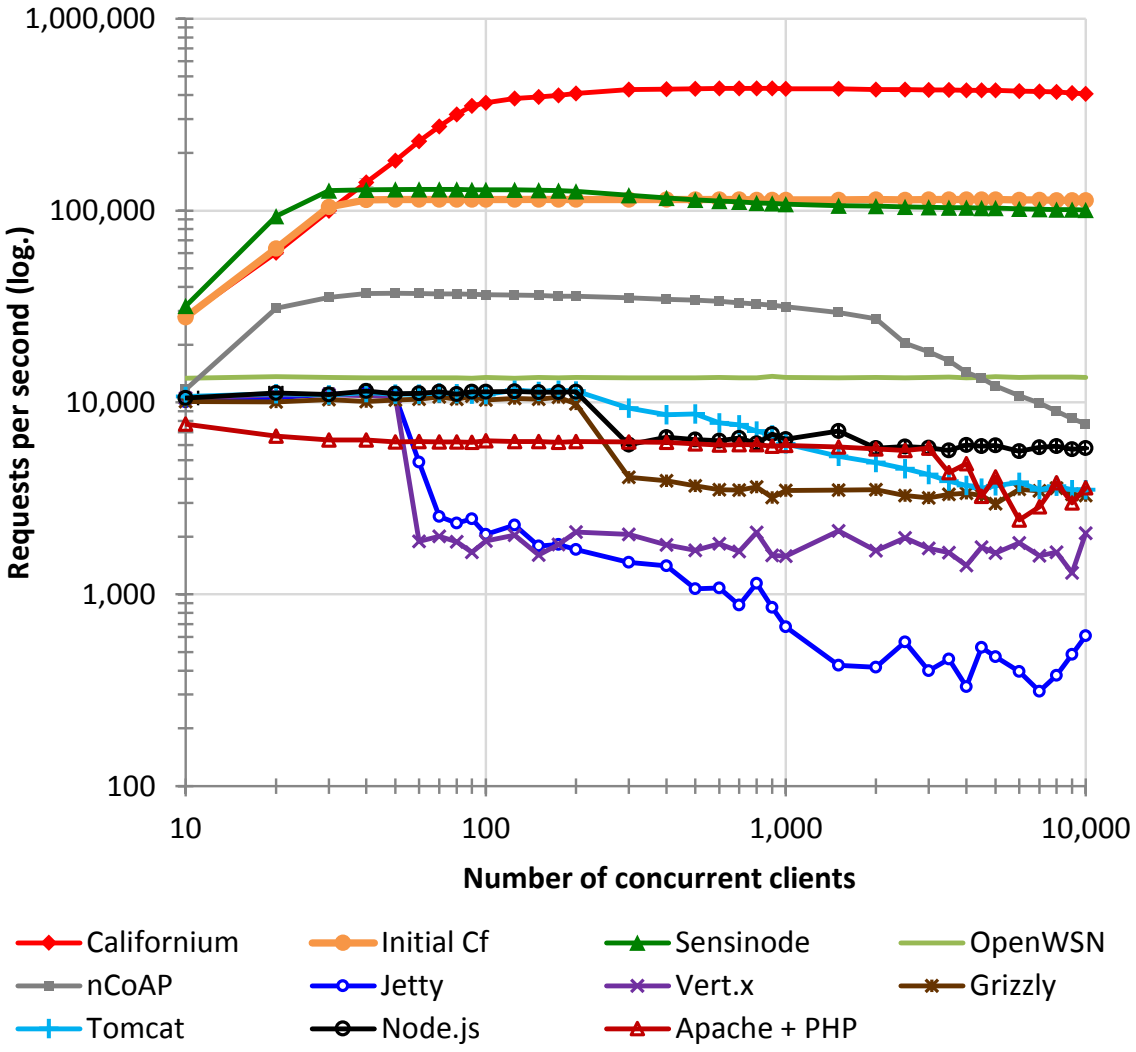


Figure 4.15: Throughput without HTTP keep-alive: HTTP servers mainly suffer from TCP overhead: the three-way handshake and slow-start mechanism. Yet short-lived message exchanges from myriads of devices is the expected traffic pattern in the IoT.

In highly concurrent service applications as found in the IoT, CoAP performs much better than HTTP-based solutions in terms of scalability. The logarithmic scale shows that our system also performs an order of magnitude better than other CoAP implementations. Compared to state-of-the-art HTTP solutions, CoAP can achieve a 33 times higher throughput for conservative concurrency factors (up to 200). For concurrency factors as expected in IoT scenarios, it even has a 64-fold increase in throughput compared to HTTP. This is mostly due to the 3-way handshake and tear-down of TCP connections, which leads to at least 9 messages to execute a single request. The overhead of the verbose HTTP headers slows the systems down further, especially considering the small payloads that are typical for IoT traffic.

4.5 Summary and Discussion

This chapter answers the research question how Web technology can be scaled up to the expected hundreds of billions of connected devices. We presented our CoAP-based system architecture for scalable IoT cloud services. Furthermore, we provided a detailed evaluation of CoAP in unconstrained environments, namely the service backend. The *Californium* architecture reflects our experience with CoAP implementations, but is also inspired by proven architectures for Web servers, as almost no CoAP backend system architectures exist so far in practice. Our system architecture is structured in three stages that are separated by queues to enable individual concurrency models. The stages also allow for flexible configurations with multiple endpoints for the business logic. We also provide a reference implementation of our architecture, the *Californium (Cf) CoAP framework*. It has become an open source project within the Eclipse Foundation and is publicly available on GitHub under EPL+EDL dual-licensing. This chapter also explained the details of our architecture on the basis of the components and structure inside our framework. In addition, we reflect on the lessons learned when implementing CoAP without resource constraints as first priority. The evaluation shows that our 3-stage architecture greatly utilizes the resources of today's multi-core systems. Our Californium reference implementation outperforms other CoAP systems with a three times higher throughput. Furthermore, it is much more scalable than popular high-performance HTTP Web servers.

As a more general result, we showed that CoAP's low overhead also has significant advantages over HTTP in the IoT service backend. With up to 64 times higher throughput than state-of-the-art Web servers, CoAP-based cloud services can handle the expected myriad of IoT devices in an efficient way. Thus, we propose to limit the use of CoAP-HTTP cross-proxies to the transitional period. In the long-run, IoT cloud services and

Web integration platforms should speak CoAP directly to be able to scale to vast numbers of concurrently connected devices. Yet when an IoT cloud service interacts with a small number of other services or a load balancer, long-lasting TCP connections using HTTP/2.0 or the upcoming CoAP-over-TCP binding are still a good choice.

There are two limitations to this work. First, we focus on the scalability of the essential networking and backend support technology to implement the vision of the IoT. Future work also needs to take security aspects into consideration: We already implemented a DTLS solution in our Scandium (Sc) project. A DTLS handshake poses similar problems as establishing a TCP connection and requires additional state to be kept on both peers. To achieve optimal security profiles for the IoT, similar experiments need to be run with CoAPS and HTTPS. It is advisable, however, to wait for the (D)TLS v1.3 specification to stabilize, since the reduction of RTTs is one of the main goals of the new revision. Second, we limited our scope to vertical scaling, that is, optimizing performance on a single machine. While our solution can already handle a large number of connected IoT devices, horizontal scaling will be needed for reliable data management and big data analytics. Once the data is available in an IoT cloud system, however, the challenges are not too different from traditional high-performance information systems and generic techniques for horizontal scaling may apply.

Chapter 5

The Human in the Loop

The concept of the IoT is to connect the real world to the virtual world, so that digital services can perform pervasive automation tasks. Primarily, automation requires machine-to-machine (M2M) communication to sense and control physical processes. Yet IoT applications ultimately pursue user-defined goals to provide added value. Thus, humans are typically in the loop, in particular, to create and customize IoT applications. In general, programming these applications is challenging because developers have to be knowledgeable in various technical domains, from low-power networking, over embedded operating systems, to distributed algorithms. Hence, it will be challenging to find enough experts to provide software for the vast number of expected devices. This also affects the economical aspects of the IoT, in particular *time to market*. To realize the IoT vision, the new technical solutions must also be implementable in products in a timely fashion to become available on the global market. Thus, we need to investigate adequate programming models and tools to foster a commercial launch of the IP-based IoT.

The previous chapters showed how the Constrained Application Protocol (CoAP) helps to solve the technical challenges for scaling Web technology down to resource-constrained devices and up to hundreds of billions of connected nodes. Yet CoAP also allows for well-known patterns from the World Wide Web to be applied in the IoT domain. In this chapter, we focus on the usability aspects for the IoT. Developers require a Web-like programming model and adequate tools to create software with the same creativity and productivity as experienced in the World Wide Web. We present the concept of a runtime system that enables Web-like scripting and portable application code for the IoT. Our RESTful runtime container exposes scripts, their configuration, and their lifecycle management through a coherent RESTful interface using CoAP. An object API similar to the XMLHttpRequest (XHR) enables direct interaction with resource-constrained IoT devices and scripts can simply export internal state as Web resources to enable rich Web-like mashups. Such a

runtime system is implemented by our *Actinium (Ac) app-server for Californium*, which we use to evaluate our concept. Furthermore, we prototype Web browser support for CoAP to complete the Web experience for the IoT. The Web browser not only represents a well-known user interface for devices, but also a widely used tool to debug and test applications during development and deployment. Our evaluation shows that the portable scripting approach is well suited for IoT business logic. A user study also reveals that the majority of IoT developers prefers Internet protocols and agrees that the patterns from the Web ease IoT application development.

This chapter is based on our publications [105] and [108]. The following section gives an overview over related work. Next, we present our Actinium runtime system in Section 5.2 and evaluate the concept in Section 5.3. Section 5.4 introduces our *Copper (Cu) CoAP user-agent*, which adds CoAP support to the Web browser. We then discuss the results from a user study on the usability of our approach in Section 5.5.

5.1 Related Work

Programming networked embedded systems is challenging and will be the biggest challenge for the IoT. One of the main incentives to use Web technology for the IoT is user-friendly programming model and the high availability of Web developers. In the following, we provide an overview how programming of resource-constrained devices evolved and what approaches are currently in use.

5.1.1 WSN Programming Models

With networked embedded systems becoming more and more complex, the device firmware is often based on an embedded OS such as *Contiki* [48] or *TinyOS* [119]. They provide a fundamental set of system tools such as timers, multi-tasking abstractions, and hardware drivers—just like classic OSs—but are specifically designed for resource-constrained platforms. In the field of WSNs, applications are traditionally programmed directly atop these OSs. The code is often written in the same language as the OS, statically linked to it, and not strictly isolated from it. This makes applications efficient, but also error-prone and complex, as programmers need to know the details of OS and platform. When the software needs to evolve, developers proceed with network-wide full image replacement [93, 164], incremental update [148], or dynamic linking [47]. Although these techniques are appealing, they turn out to be difficult to use in real deployments because of

bad network connectivity and implementation bugs [113, 194]. Furthermore, the developer must be knowledgeable in embedded programming and the particularities of the OS.

An interesting alternative is to embed a virtual machine (VM) in the device and deploy applications compiled as bytecode. This approach provides a higher-level of abstraction, provides dynamic loading, software isolation, and minimizes the size of compiled applications. One of the first VMs for sensor networks was *Maté* [118], a framework for domain-specific VMs with mobile code. Later, general-purpose VMs have been developed for resource-constrained platforms. *Darjeeling* [31], for instance, supports a subset of the Java language and even a garbage collector. There are also commercial VM solutions such as the PreonVM[1] by Virtenio. While the software development is easier and the code more portable, the virtual machines also have higher platform requirements.

Macro-programming is a solution that aims for programming a network as a whole by providing network-scale abstractions. *Abstract regions* [193], for instance, provide shared variables among neighboring nodes to allow for efficient state reductions, spatial operations, and trade-offs between energy consumption and sensing accuracy. In analogy to MatLab, *MacroLab* [87] uses vector programming abstractions to operate on the network as a whole. This eases finding nodes with specific sensor readings and aggregating the data. *TinyDB* [126] provides a SQL-like database abstraction to the user to perform sensor readings. A similar approach is implemented by *Cougar* [201]. They provide in-network processing, where data is aggregated along the hops with convergecast messages. This is, however, not practical when multiple stakeholders are involved, as the mechanism can only provide averages, maxima, etc. of the sensor readings. In contrast, *Nano-CF* [82] is designed for concurrent applications on a WSN infrastructure. It optimizes the execution and traffic of a predefined number of tasks on the motes through *rate harmonized scheduling* and packet aggregation, or concatenation when aggregation is impractical. *Nano-CF* also provides code dissemination to create and update tasks at runtime. Macro-programming solutions focus on networks of homogeneous devices, though.

5.1.2 Toward Web-like Programming Models

The early *Marionette* system [196] implements uniform interfaces directly on heterogeneous sensors. It treats sensors as simple servers that are connected to a powerful client, which takes care of the computation tasks. While still relying on proprietary messages, Marionette was a first step toward a ubiquitous infrastructure as we now have with CoAP. A further step toward Web-like interaction was presented with *sMAP* [41], which uses

[1] http://www.virtenio.com/en/products/virtual-machine.html (accessed on 12 Feb 2015)

a custom JSON data model over an embedded, binary version of HTTP (EBHTTP) to integrate various sensors, building automation systems, and traditional Web services. The *Devices Profile for Web Services* (DPWS) can bring a Web application standard to low-end devices. With an efficient binding [137] and EXI compression [35], devices can directly process SOAP messages, even on mote-class platforms. Usually, the EXI handler code is generated for each application to enable on-the-fly processing of the SOAP document and small memory footprints [117].

Scripting languages raise the level of abstraction even higher, as they provide the programmer with the ability to batch well-defined basic operations. They increase productivity by making applications self-contained, focused on functionality, and easy to test interactively [145]. Scripting is particularly well-suited for the IoT, as the functionality of connected devices is built upon a simple set of actions: periodic sensing, alarm triggering, and actuation. On-device script interpretation, as performed by *SensorWare* [26] or *dinam-mite* [76], has comparatively high system requirements. Thus, scripts are usually compiled before sending them to the devices. This is done by *EcoCast* [184], which also links the code incrementally and patches the devices efficiently. There are also innovative commercial products such as Snap [2], which also builds on Python to create IoT applications.

5.1.3 The Web of Things

The Web of Things (WoT) initiative [81, 197] advocates to bring the full Web experience to the IoT. It aims at out-of-the-box interoperability for sensing and actuation devices and reusability of available systems by using HTTP and RESTful interfaces. Unless devices are powerful enough to run a full TCP/IP stack with HTTP, they have been integrated through application-level gateways that host a Web server and translate to classic proprietary device protocols [182]. Here, one of the first works is *pREST* [43], a REST-based protocol for an ubiquitous computing system that provides access to sensors through an HTTP server on a computer gateway. The gateways, however, only abstract from the networked embedded systems, which still have to be programmed in the traditional way.

The overall goal of the WoT initiative is to create a universal application layer and ecosystem for the IoT based on the World Wide Web. For this, it promotes the use of Web scripting languages, which enable *physical mashups* of IoT devices [80]. The community has been developing several toolkits and cloud services that provide APIs to access sensors and actuators. The *WebPlug* framework [144], for instance, is a Web mashup software that connects HTTP-enabled devices. It provides a clever URI syntax to access collections, histories, etc., so that URIs become first class citizens of the programming

languages. The platform by Boussard et al. [27] combines a virtual object abstraction with the Web Ontology Language (OWL) to enable semantic-driven applications for smart spaces. *WoTKit* [19] is a recent toolkit that unifies several requirements for IoT cloud platforms such as an easy-to-use API, descriptions, a processing engine, sharing, and visualizations. The *COMPOSE API* [152] is the product of a recent EU project that aims at an open marketplace for IoT objects, services, and applications. It has a resource-oriented architecture (ROA) but does not follow REST, since URIs are statically defined and consist of centrally defined identifiers.

Through the standardization of CoAP, the idea of physical mashups [80] becomes even more interesting, as IoT devices and can be integrated seamlessly by directly mashing up their Web resources. Here, the thin server architecture presented in Chapter 3 plays a significant role, as it enables an application-agnostic infrastructure of devices that connect to the physical world [110].

5.1.4 Debugging, Testing, and User Interaction

Considering the human in the loop, the interaction with devices is closely related to the programming model, as it is integral part of debugging and testing distributed software such as IoT applications. Early WSN systems were not designed for direct interaction and require custom software that sends the corresponding messages for interaction through the sink node [18, 128, 195]. With the introduction of macro-programming and scripting, solutions started to provide system-specific graphical user interfaces (GUIs) or shell-like interface. Examples are the TinyDB GUI to construct and execute queries [126] or Marionette's support for the Python shell [196]. Typically, these solutions are still bound to homogeneous devices, though.

In the WoT, the interaction with individual devices has become more central and can easily be provided through Web frontends. These can be custom made Web pages or dashboard components of WoT platforms that can be configured to reflect the device properties [19, 167]. By using semantic descriptions of device resources, such frontends can also be generated for different platforms. Mayer et al., for instance, propose a model-based interface description scheme and present an automatic user interface generation for the Android OS [131]. The *DPWS Explorer*[2] is a standalone application that uses the Devices Profile for Web Services (DPWS) description to provide a GUI for corresponding devices. Such device browsers were already proposed for physical artifacts that are linked to virtual counterparts through RFID tags or barcodes [163].

[2] http://ws4d.e-technik.uni-rostock.de/dpws-explorer/ (accessed on 12 Feb 2015)

Our work aims at a generic tool that supports software developers already during the programming, debugging, and testing phases. For RESTful Web services, and hence the WoT, the Web browser has become the primary development tool. All modern browsers are shipped with development tools[3] [4] [5] [6] and can be further enhanced through add-ons such as RESTClient[7]. Consequently, we promote direct CoAP support in the Web browser to better integrate device interaction into the Web ecosystem. In the following, we present our concepts and tools to foster a user-friendly, Web-like programming model for the IoT.

5.2 RESTful Runtime Containers for the IoT

The IoT needs new software concepts and architectures that are different from traditional networked embedded devices. The latter have mostly been independent silos that fulfill specialized tasks. With the protocols presented in the previous chapters, this situation has changed. However, even with the rising presence of light-weight IP stacks, programming IoT applications remains unnecessarily difficult. Developers have to program different operating systems, have to focus on platform-dependent issues, and have to design detailed network interactions. For the IoT to take off, its programming model needs to be as easy as scripting Web applications. Furthermore, it requires an interoperable ecosystem as promoted by the WoT initiative.

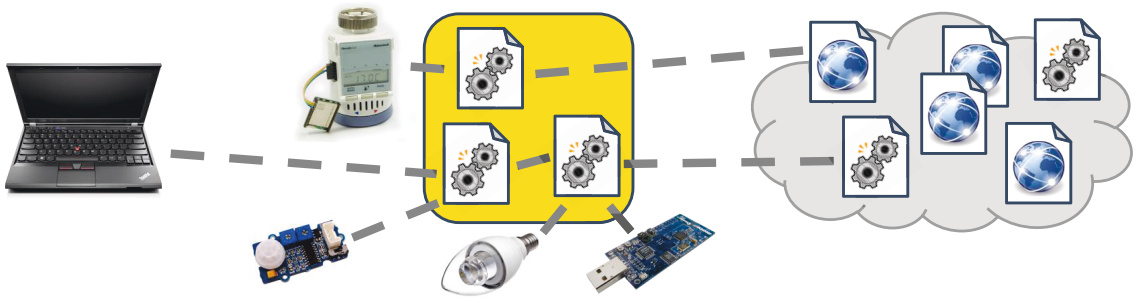


Figure 5.1: A scripting language can enable user-friendy mashups of resources hosted directly on IoT devices as well as other scripts or remote Web services. Our runtime concept also performs dynamic installation, updates, monitoring, and removal of scripted applications through RESTful interfaces.

[3] https://developer.chrome.com/devtools (accessed on 12 Feb 2015)
[4] https://developer.mozilla.org/en/docs/Tools (accessed on 12 Feb 2015)
[5] msdn.microsoft.com/library/ie/bg182326 (accessed on 12 Feb 2015)
[6] https://developer.apple.com/safari/tools/ (accessed on 12 Feb 2015)
[7] https://addons.mozilla.org/firefox/addon/restclient/ (accessed on 12 Feb 2015)

With our *Actinium* runtime container, we present the concept for a RESTful runtime system that enables scriptable, portable, and interoperable IoT applications that are composed of modular 'apps.' In general, runtime environments for the IoT need to address the following requirements:

1. *Full support for IoT devices*: Applications must be able to integrate resource-constrained IoT devices in a transparent way. Hence, there must be an adequate API that treats Web resources of devices similarly to classic Web resource.

2. *RESTful interactions*: The REST architectural style enables a loose coupling between the application endpoints, which makes Web mashups robust to changes in the ecosystem. Clients and servers, and hence services, are able to evolve independently without breaking the application. Thus, all interactions must be channeled through RESTful interfaces.

3. *Mobile code*: We need a runtime system similar to the Web browser that provides a common API so that the application code, a script, is not bound to a specific system. This is important for the virtualization in IoT cloud services, but also for resource-constrained devices. The latter sometimes have too little power to perform all the processing necessary for the services they offer. Since they are Web servers, however, they can leverage the *code-on-demand* constraint of REST and delegate the processing to an associated runtime system.

4. *Headless operation*: Unlike the Web browser, the runtime system will only need a minimal user interface, since the main objective is to run IoT automation tasks in the background. Whenever user interaction is required, the WoT allows for the seamless integration of Web platforms, which can provide the necessary visualizations.

5. *Multitenancy*: The runtime system must be able to host apps from multiple client-organizations in parallel. They may not interfere with each other unless intended by the application. For this, they must run in separation and provide the necessary interfaces to exchange internal states in a secure way.

6. *Lifecycle management*: With mobile IoT devices, changing requirements, and dynamic service bindings, application code becomes volatile. To ensure robust execution of tasks, the runtime system must provide full lifecycle management support to dynamically install, start, update, and stop the scripts.

7. *Security*: Due to the privacy and safety implications of the IoT, the runtime system must provide security mechanisms that allow for adequate policies. Here, tool support plays an important role, since wrong or missing configuration of the cryptographic protocols by the users is a main source of security breaches.

Instead of defining a new language, we use the most widespread scripting language in the Web: JavaScript. We enhance the scripting environment that is well-known from the Web browser with a CoAP API to communicate with resource-constrained devices. Our reference implementation, the *Actinium (Ac) App-server for Californium*, is part of the Eclipse Californium project and publicly available on GitHub.[8]

5.2.1 Runtime Container Design

We propose an architecture for networked embedded systems, where IoT applications are realized through scripts running in a cloud or fog service. They can access the elementary functionality of IoT devices through their RESTful interfaces and combine them with classic Web services. This approach offers a great deal of flexibility and scalability because IoT applications become computer-hosted *apps* rather than embedded software. In Actinium, we extend the WoT approach and advocate a truly end-to-end RESTful approach where not only the devices have RESTful interfaces, but the runtime container itself is RESTful. This concept further allows applications to be shared through an 'appstore,' that is, they can easily be uploaded, downloaded, customized, signed, etc. We use the JavaScript scripting language because it is the most prominent language for Web mashups and well-known by many application developers and even by end-users. The concept, however, is not limited to JavaScript and can also be transferred to other languages, for instance, by extending the polyglot Vert.x framework. While we focus on the communication over CoAP to specifically address resource-constrained devices, the concept also applies to HTTP-based devices and services. In analogy to the GUI elements of a browser, which are focused on user interaction with event-handlers like `onclick`, our runtime container provides IoT-specific elements such as resources that have `onget` and `onpost` handlers, and an `app` object API instead of `window` for facilities such as `app.setTimeout()` or `app.getNanoTime()`.

Apps as Resources

To implement IoT applications, we use modular scripts, which are designed as resources to fully leverage the RESTful paradigm. They can provide their results through GET handlers, accept stimuli by POST, or can be configured via PUT. They can be mashed up together with devices and other application modules. Hence, application logic can be distributed over the complete network: in the cloud for public and scalable applications, locally (in the fog) for closed or time-critical applications, or directly in the LLN when

[8] https://github.com/eclipse/californium.actinium (accessed on 12 Feb 2015)

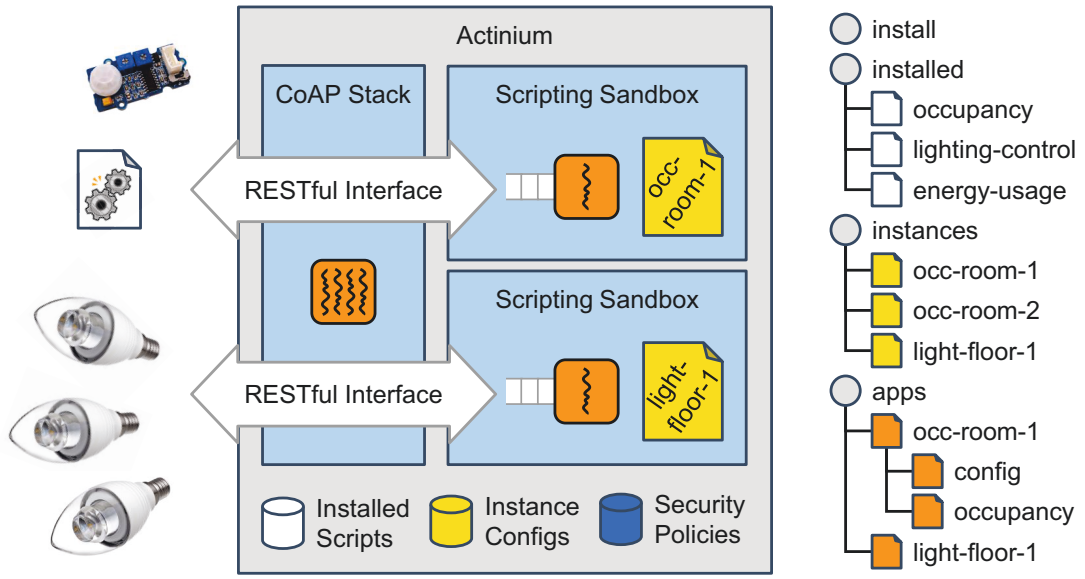


Figure 5.2: An overview of our system architecture: Actinium apps are executed in their own sandbox. They only communicate through their RESTful interfaces: with IoT devices, other apps, and other servers on the Web. The tree on the right shows the available resource structure when three scripts are installed, two are instantiated (one of them twice), and two are running.

more powerful IoT devices are available. The central API for this is the `app.root` object, which represents the root resource of a running script:

```
1 // a handler for GET requests to "/"
2 app.root.onget = function(request) {
3     // that returns CoAP's "2.05 Content" with payload
4     request.respond(2.05, "Hello world");
5   };
```

Listing 8: Actinium resource API

We limit the scripts to consist of a single file, which forces developers to break complex applications down into multiple scripts that have to communicate through their RESTful interfaces. These modules then become reusable by other applications, which renders the mashup concept more powerful. Multiple motion sensors, for instance, could be wrapped by one occupancy app per room, filtering the sensor data and providing a boolean value as output. A lighting control app then combines these occupancy resources with the control resources of the lighting system of each room and automates them for energy savings. Such a chain of apps could be continued by an energy usage app for example. Actinium apps can also have sub-resources to provide a hierarchical structure for the exported data:

```
var threshold = 0;

// a sub-resource "/config"
var sub1 = new AppResource("config");
// that accepts PUT requests
sub1.onput = function(request) {
    // to configure the threshold
    threshold = request.payloadText;
  };
app.root.add(sub1);

// a sub-resource "/occupancy"
var sub2 = new AppResource("occupancy");
// that returns true or false depending on a given value
sub2.onget = function(request) {
    request.respond(2.05, value > threshold ? "true" : "false");
  };
app.root.add(sub2);
```

Listing 9: Actinium sub-resource concept

Mashups

The idea of Web mashups is to combine different Web services to provide a service of higher value. Mashups are light-weight applications that are easy to create, provide flexible solutions, and generally leverage the high productivity of scripting. These are properties that are ultimately required for the IoT, where each user is associated to a plethora of heterogeneous devices. Due to the connection to the physical world, powerful applications can already be created by just combining and evaluating sensor or status information, and instantaneously triggering events for actuation or storing the results for big data analytics.

To mash devices up, apps must take the client role. A WebSockets-like API would be an option, but it is for arbitrary data traffic and does not follow a RESTful design. We provide the *CoapRequest* object API, which is designed similar to the XMLHttpRequest (XHR) object API [98] of Asynchronous JavaScript and XML (AJAX):

```
var req = new CoapRequest();

// request the PIR sensor resource of a mote via CoAP
req.open("GET", "coap://mote1.example.com/sensors/pir", false /*synchronous*/);

// with a application/json response
req.setRequestHeader("Accept", "application/json");
req.send();  // blocking

// and log it to the console after send() returns
app.dump(req.responseText);
```

Listing 10: Actinium CoapRequest object

Due to the resource design of Actinium apps, they are able not only to mash up services of devices, but also of other apps using the same interface. Our runtime system also supports the normal XHR to include traditional Web services in the mashups. The following snippet contacts a default Contiki border router, which usually hosts an HTTP Web server to list the available routes to connected nodes:

```
var xhr = new XMLHttpRequest();

// GET "/" with a list of all LLN neighbors and routes
xhr.open("GET", "http://br.example.com/", false);
xhr.send();

// and retrieve all LLN node addresses via regular expressions
var addresses = xhr.responseText.match(/[0-9a-z\:]+(?=\/128)/g);
```

Listing 11: Actinium XHR support

Both request objects also support asynchronous communication. Thus, an app can also send multiple requests in parallel, which enables interleaving of long-lasting requests to a group of nodes. The responses are then handled by a callback function that implements `onload`. A distinctive feature of CoAP are unreliable requests, which can simplify continuous polling. To choose between CON and NON messages, an additional boolean is passed to `open`, while the default value is `true` for CON requests:

```
// define the callback of an existing CoapRequest
req.onload = function() {
    if (this.responseText=="false") switchOffLights();
  };
// and a timeout with a timeout callback
req.timeout = 5000; // in ms
req.ontimeout = function() {
    app.dump("Request timed out!"); // to console
  };
// and send the an asynchronous, non-confirmable request
req.open("GET", "coap://app-server.example.com/
         running/occ-room1/occupancy", // other app
         true /*asynchronous*/, false /*non-confirmable*/);
req.send(); // non-blocking
// and continue execution immediately
```

Listing 12: Actinium asynchronous CoapRequest calls

A key feature of CoAP for IoT applications is *observing* resources [84], which is initiated through the `Observe` option. These native push notifications can be used similarly to HTTP's chunked transfer (streaming) in AJAX. Our CoapRequest object uses the `onprogress` callback to inform the apps every time an update is received:

```
var req = new CoapRequest();

// define the callback for notifications
req.onprogress = function() {
    // only contains payload of last message (unlike XHR)
    update(this.responseText);
  };

// request is DONE, i.e., the observe relationship ended
req.onload = function() {
    app.dump("Observing terminated"); // to console
  };

req.open("GET", "coap://mote1.example.com/sensors/pir",
         true /*asynchronous*/, true /*confirmable*/);
req.setRequestHeader("Observe", 0);
req.send(); // non-blocking
```

Listing 13: Actinium CoapRequest observe support

RESTful Lifecycle Management

Our design allows for dynamic installation, updates, and removal of scripts in a RESTful manner as depicted in Figure 5.3: A resource, for instance `/install`, accepts POSTed scripts, adds and stores them in its resource tree, e.g., under `/installed`, and reports back the new `Location` path via the corresponding CoAP option. The same Actinium app might be required several times, for instance on a per-floor or per-room basis. Thus, we distinguish between installed apps, which represent the code, and their *instances*, which hold individual configurations. They are created by POSTing the configuration

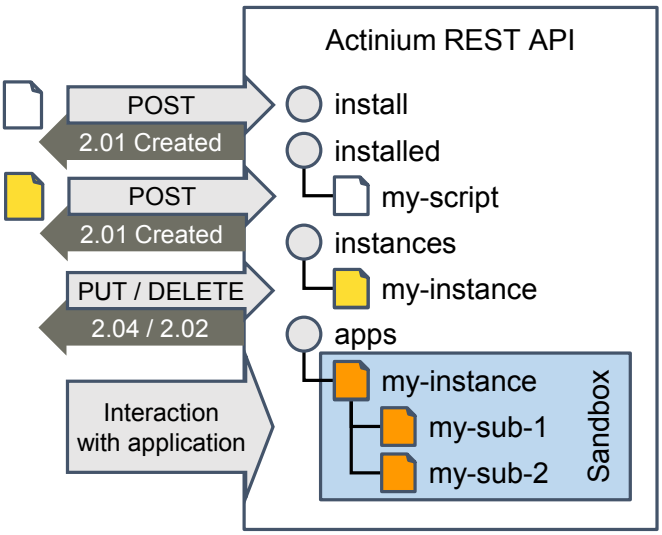


Figure 5.3: The lifecycle management is RESTful as well.

to the intended script, in our example under `/installed/<app>`. These instances have two lifecycle states: stopped and running. In the latter case, they are available under `/apps/` with their instance name. The full URI of the previous occupancy example would thus look like `coap://actinium.example.com/apps/occ-room1/occupancy`. Overall, this means that scripts that run in such a runtime system can be used to install new scripts and to manage the instances. In a production system, this must underlie access control, just like any user interface for lifecycle management.

5.2.2 Security Considerations

The IoT is about connecting the virtual world to the physical world. As such, it requires security at different levels: First, applications from different providers running for different users in the same runtime environment, must not leak sensitive information or be harmful to one another. Second, communication with the devices must be secure, so that sensitive information is authenticated, exchanged confidentially, and integrity protected. This section reviews how we handle these security issues in our design.

Securing Application Execution

As with operating systems, users have to fully trust the application server. That given, trust into individual applications can be relaxed by sandboxing and enforcing certain policies. To this end, our concept provides three mechanisms:

Isolated Apps It is crucial that there is no unintended interference between applications. In particular, errors occurring inside one Actinium app must not influence the behavior of other apps. Our runtime system addresses this requirement by executing each app within a separate sandbox. The only way for the scripts to communicate with each other is through their RESTful interfaces, either locally or over the network.

Policies Keeping apps in a sandbox also enables the container to have strong control over them. By default, there are no restrictions in terms of when and how they are allowed to access other resources or to be accessed. An end-user, however, might want to define such restrictions, for instance prohibit activation of the television during night or disallowing apps to access local cameras. Our container allows for the definition of such boundaries and guarantees compliance. By providing read-only access to the policies, applications can use defensive programming to avoid relentless trials and crashes.

Monitoring Finally, the sandbox wrapper eases the monitoring of the scripts. The runtime container records traffic statistics including the amount of transferred data. In addition, it can monitor the CPU time specific threads consume. If the runtime container stresses the CPU, end-users can check which app causes the high load. Furthermore, they can check whether the runtime system is congested by many incoming requests or actual misbehavior and only stop the script if needed.

Securing Communication

It is important that only authorized applications and users interact with a given device to prevent privacy invasion, or worse, malicious actions with unpredictable impact on the physical world. Following the open standards, we base our security architecture on the DTLS protocol [161], an adaptation of TLS for UDP, which provides CoAP with end-to-end integrity, authentication, and confidentiality. There are two aspects for integrating this security model into our architecture:

Authenticating Applications We argue that the traditional model of the Internet, where applications (e.g., Web sites or smartphone apps) are only signed by the providers, is not suitable for the IoT. Often, authenticated applications may access any device and users do not want to grant access to their things based on the application designers' choices. Thus, we propose a model in which the users sign each *configured instance* of an application instead of providers signing the code. For each app that is deployed by the user, a key pair is generated to produce an *app certificate*, which is signed by the user.

Authorizing Applications Signed apps should not be allowed to access any device, since some are more critical than others and interference may cause safety hazards. Instead, each app must be authorized per device, or even better per resource. This can be done by installing a raw public key of authorized apps on the device. This procedure can be automated through a configuration tool, which can be part of the runtime container.

Providing Data Integrity and Confidentiality Another concern in the IoT is to check the integrity of data originating from devices and to guarantee confidentiality while transporting it. DTLS, like TLS, allows authentication of both parties during the session establishment. By using this feature, an application can be guaranteed about the identity of the data source, and can send and receive data confidentially through any network. Experiences from TLS, however, show that proper, user-friendly tool support is required to make a certificate-based security model work. Thus, corresponding mechanisms should be provided with the runtime container.

5.2.3 Actinium (Ac) Implementation

The *Actinium (Ac) app-server* is implemented as an add-on for our Californium framework. It already provides the necessary concurrency model, since each Web resource can be configured with its own thread pool. Sub-resources that do not have a thread pool will automatically use the one of their parent or transitive ancestor. By choosing a thread pool size of one for the app base resources, the resource handlers become single-threaded, which creates the event-driven execution model of JavaScript. Similar to the Web browser, the script is executed once when loaded and it can initiate actions by posting events to the queue of the thread pool. This is usually done by recursively setting timeouts with a corresponding callback function or by defining callback for external events. To implement the JavaScript interpreter, we use the Mozilla *Rhino* project[9], as we were bound to Java 6 when conducting this work.

AJAX's XHR is not part of ECMAScript [57] and hence not supported by Rhino. We use the *E4XUtils* extension library for ECMAScript for XML (E4X) available from IBM[10] to include this API. Our *CoapRequest* object API is backed up by custom Java code that wraps the client functionality of Californium. The new working draft of the *XMLHttpRequest Level 2* specification states that "some implementations support protocols in addition to HTTP and HTTPS" [189]. As the functionality, however, slightly differs from CoAP and the name of the object would become even more confusing, we decided for separate APIs. Once push notifications are solved for HTTP in a common way, it will be possible to change over to a unifying API that better suites the REST abstraction and is free of the XML legacy.

As complex applications shall be built by mashing up other apps, there is no container format such as a WAR-like archive. Each app is a text-based script with an `application/javascript` Internet Media Type, which is persisted in a single file and loaded into a wrapper object for execution. The wrappers also implement the sandboxing described in the previous section.

The secure communication is provided by DTLS 1.2. It is implemented in *Scandium (Sc)*, which is part of the Eclipse Californium project[11]. The implementation of the security tool support is left open in this work. It heavily depends on the results of the recently chartered IETF working group for Authentication and Authorization for Constrained Environments (ACE).Authentication and Authorization for Constrained Environments (ACE).[12]

[9] http://www.mozilla.org/rhino/
[10] http://www.ibm.com/developerworks/webservices/library/ws-ajax1/
[11] https://github.com/eclipse/californium.scandium (accessed on 12 Feb 2015)
[12] https://datatracker.ietf.org/wg/ace/charter/ (accessed on 12 Feb 2015)

5.3 Evaluation and Results

The goal of the Actinium project is to show that scriptable mashups are feasible for the IoT. In the World Wide Web, this loosely-coupled service composition allows for robust and flexible applications across different organizations. Furthermore, scripting improves productivity [145]. Actinium is publicly available as proof of concept. In this section, we evaluate the performance of the scripting approach for IoT applications.

5.3.1 Setup

We evaluate Actinium in a real-world setting with a 6LoWPAN sensor node testbed connected directly over IPv6. The *app server* is running on a 64-bit Windows 7 Workstation with an Intel Core2 Q9400 at 2.66 GHz, 8 GiB of RAM, and JavaSE-1.6. The network configuration, which utilizes our campus IPv6 infrastructure, is depicted in Figure 5.4. To be able to easily sniff the transit traffic, the border router is connected to a laptop for the experiments. The laptop has no influence on the experiments and could easily be replaced by an embedded platform with Ethernet interface such as the BeagleBone[13].

For the LLN, we use the Contiki 2.6 release[14] with the *rpl-border-router* running on a Tmote Sky (see Section 2.1) with DMA enabled for the serial line and the *Erbium server* [107] on Econotags[15]. As we focus on the app server performance, we configured the LLN with a best case scenario for applications: no radio duty cycling (RDC) for minimal latency. In a real-world deployment, latency is usually traded for lower energy consumption to enable longer battery life times through sleeping. An energy evaluation of CoAP over an RDC layer can be found in Chapter 3. Yet the LLN underlies realistic Wi-Fi interference, as we used 802.15.4 channel 21 with several surrounding access points on channels 1, 5, 9, and 13.

[13] http://beagleboard.org/bone (accessed on 12 Feb 2015)
[14] http://www.contiki-os.org/ (accessed on 12 Feb 2015)
[15] https://github.com/malvira/libmc1322x/wiki/hardware#econotag (accessed on 12 Feb 2015)

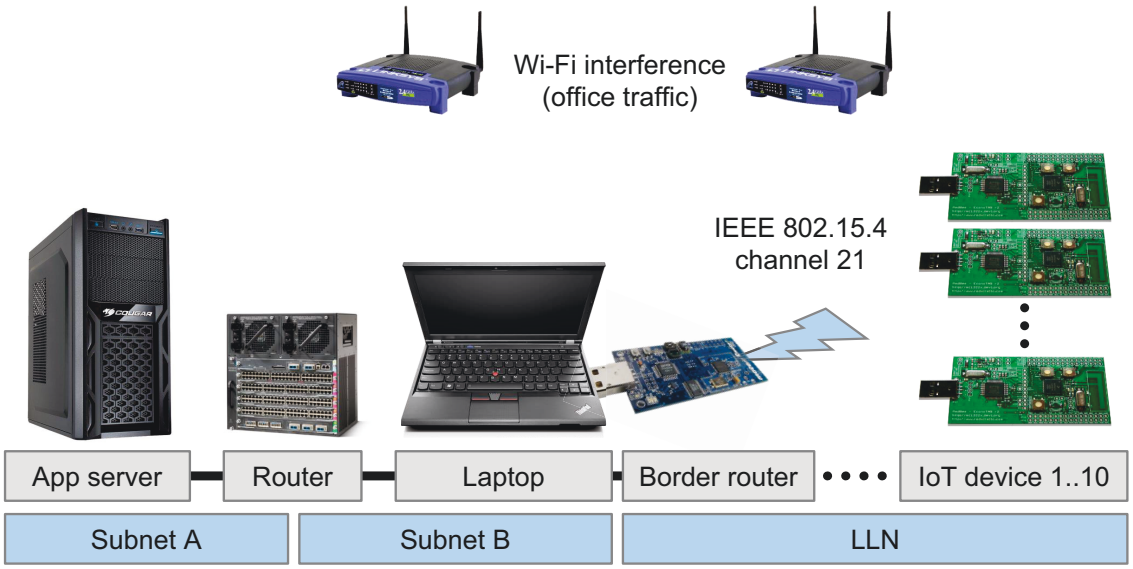


Figure 5.4: The border router has a distance of three hops to the app server. As the LLN is the bottleneck, we configured it with a best case scenario for applications: a single-hop star topology without RDC.

5.3.2 Latency Baseline

We evaluate the latency overhead that is introduced by the Rhino scripting environment and our *CoapRequest* abstraction. For this, we compare the round-trip delay time (RTT) of a JavaScript request to a native request in Java and the network latency measured with Windows's ping tool. Each measurement uses a IPv6 packet size of 80 bytes and was repeated 1000 times to account for noise, in particular for JavaScript. The results summarized in Table 5.1 only show the latency for the apps in client role, but the resource handlers for the server role have the same properties through the reciprocity discussed in Chapter 4.

The hardware that is currently available to run the border router poses a major bottle when communicating with an LLN. The radio interface is connected over a slow serial line that already induces an average RTT of 37 ms The average CoAP RTT when including a single LLN hop behind the border router is already 46 ms. Thus, the average overhead of 1.282 ms added by the scripting environment is acceptable, especially when considering that battery-powered IoT devices will employ an RDC mechanism, which increases the underlying network latency further. LLNs usually aim for an idle duty cycle (i.e., idle listening only without transmissions or interference) well below 1%. ContikiMAC, for instance, achieves 0.6% with a channel check rate of 8 Hz, which can add an extra of up to 2·125 ms = 250 ms to the RTT for a single hop.

Table 5.1: Baseline Timings: The timings were measured over 4 hops (1 LLN hop) with 1000 requests per measured system (note that Windows ping only provides a 1 ms resolution). The smaller minimum for the Actinium RTT is caused by random effects along the stack such as the IEEE 802.15.4 CSMA backoff and the Contiki scheduler.

	Minimum	*Maximum*	*Average*	*Overhead*
Ping to border router	16 ms	62 ms	37 ms	—
Ping to node	32 ms	77 ms	46 ms	+9 ms
CoAP RTT	34.173 ms	77.587 ms	47.650 ms	+2 ms
Actinium RTT	32.901 ms	97.088 ms	48.932 ms	+1 ms

5.3.3 REST Handler Performance

With an acceptable network overhead for the scripting abstractions, only the performance of JavaScript could become a showstopper. Thus, we evaluate the execution times of *Actinium*'s REST handlers with measurements that directly continue from the baselines identified in the last sub-section. We compare the Rhino JavaScript runtime of Actinium to a native Californium handler in Java and the runtime of node.js[16]. The latter is a platform that enables server-sided JavaScript for HTTP-based applications based on Google's V8 JavaScript engine.

To assess different aspects, we use three different benchmarks that are also included in the Actinium GitHub repository. Each one is implemented as request handler and measures the execution time only, that means without the network overhead assessed in the previous experiment. The input parameters are chosen so that the different complexity classes, $O(2^n)$, $O(n \cdot \log(n))$, and $O(n)$, respectively, result in reasonable execution times of up to 1.5 seconds per run.

[16] http://nodejs.org/

Fibonacci Benchmark

The Recursive Fibonacci algorithm has a complexity of $O(2^n)$ and causes a large number of function calls. It shows how efficiently the runtime systems manage deeply nested function calls. Figure 5.5 shows the outcome as intuitively assumed: Java is 3.13 times faster than node.js, which is already 3.76 times faster than Actinium. Since the C++-based V8 runtime system is more powerful than Rhino, it executes JavaScript more efficiently.

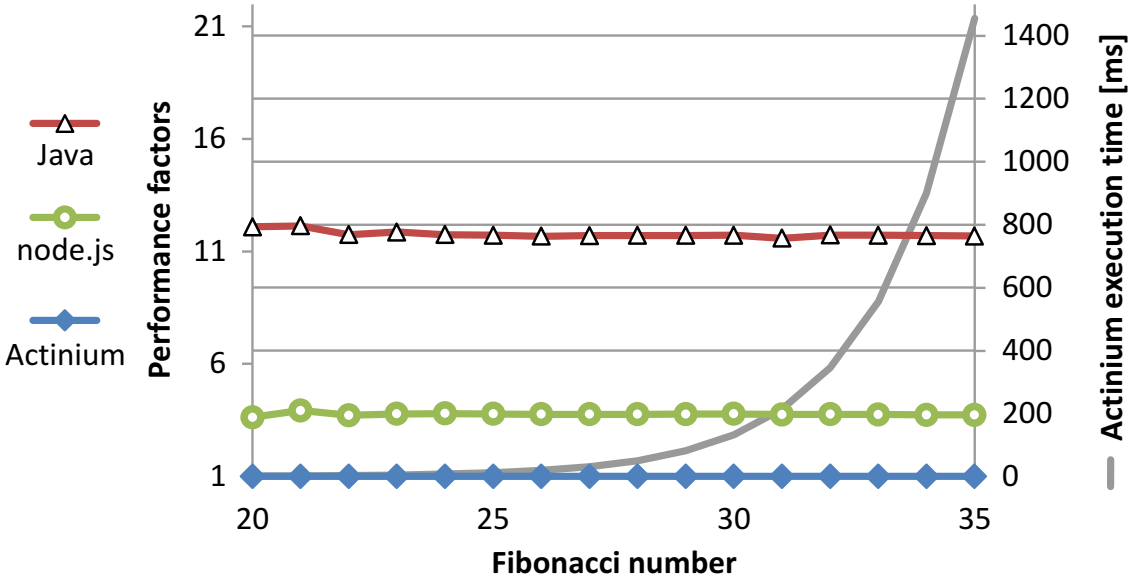


Figure 5.5: The Fibonacci algorithm over different function parameters: The left y-axis shows the performance factors between the three systems. Rhino performs on average 11.8 times slower than Java and about 3.8 times slower than node.js. The y-axis on the right indicates Actinium's absolute timings on our test system. We only show these, as the curves look qualitatively the same for all three runtime systems.

Quicksort Benchmark

The next benchmark sorts an array of double-precision floating point numbers using the Quicksort algorithm, which shows how efficient the runtime systems handle memory access. Unlike the Fibonacci benchmark, the performance factors are not virtually constant over the input parameters. Compared to Java's average speed-up of 18.2, both JavaScript runtime systems degrade with increasing array sizes. node.js scales a little worse, but on average it still performs 7.13 times better than Rhino as shown in Figure 5.6.

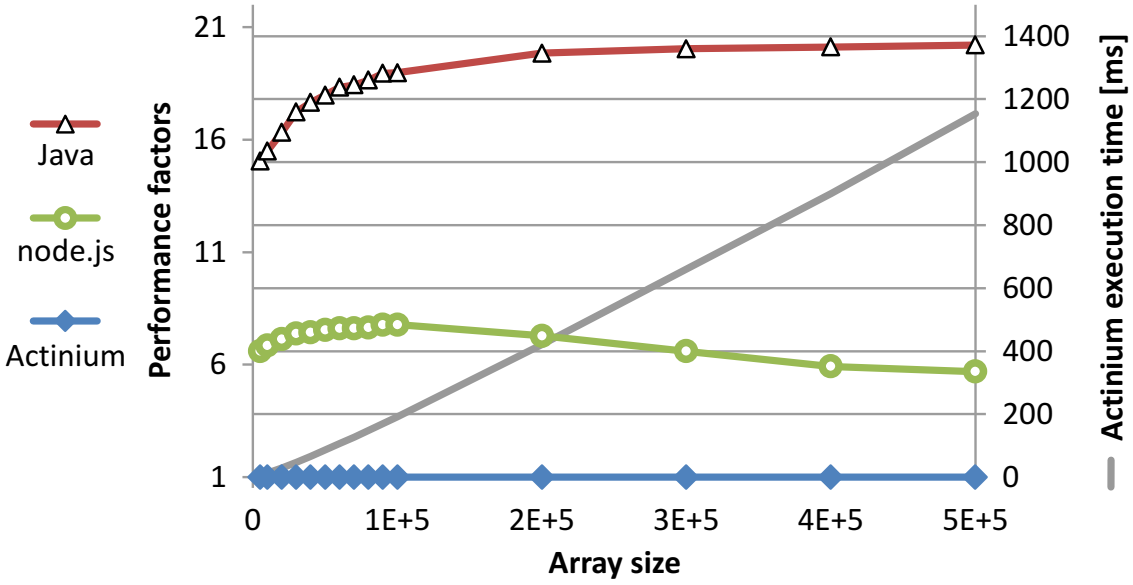


Figure 5.6: The Quicksort algorithm over varying array sizes: For memory-access-intensive tasks, Java performs best with the largest overall speed-up. For this benchmark, the speed-up factors vary and Java even gains performance while node.js slightly degrades when the arrays become very large.

Newton Square Roots Benchmark

Newton's Square Root is a fixed-point algorithm that iteratively computes the square root for a number. Since the result does not matter, we arbitrarily define eight iterations and vary the number of calculated roots. With this algorithm, we compare how efficiently the runtime systems execute arithmetic operations. In Figure 5.7, the speed-up factors of 3.25 for Java and 4.14 for node.js are close together and clearly show the strength of scripting for this kind of computation.

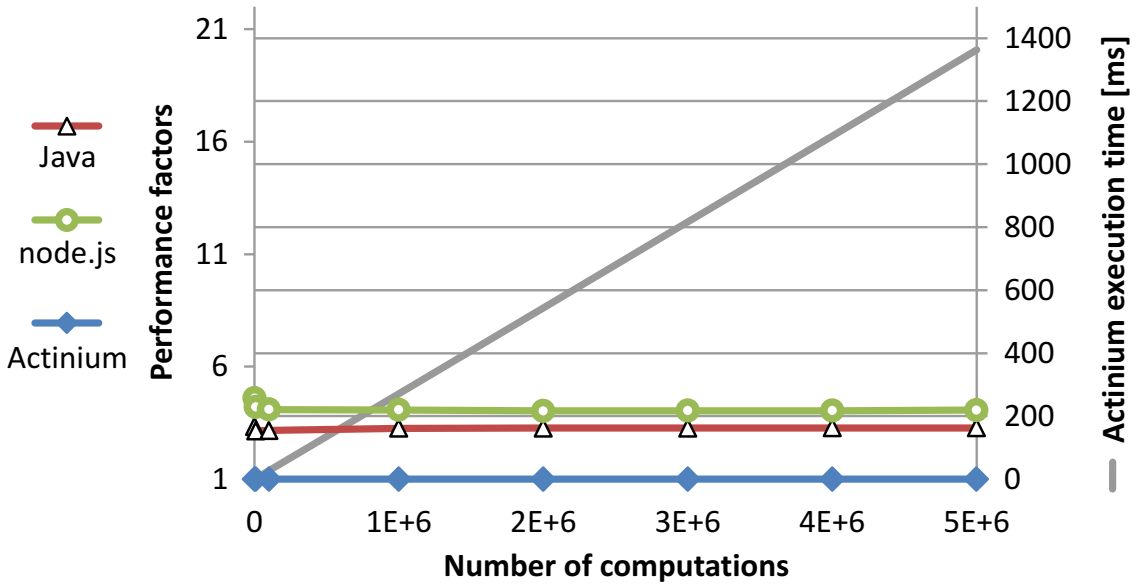


Figure 5.7: The Newton algorithm over a growing set of processed numbers: As Newton's method has a steady linear growth rate, less measurements were taken. An interesting result is to confirm that JavaScript has its strengths in pure computations and node.js even outperforms Java.

5.3.4 Multitenancy Performance

Actinium supports multiple apps running and communicating at the same time by multiplexing CoAP incoming and outgoing requests based on the URI and the token, respectively. Spawning many apps in parallel is not a problem for the underlying Californium framework. Thus, we have to investigate the network traffic to reason about bounds for concurrently running scripts. The bottleneck lies in the bandwidth of the destination LLN, which can become congested with too many messages. We conduct an experiment where ten asynchronous requests are sent to ten different nodes in one LLN in parallel. We measure the overall response time, that is, the RTT between the first outgoing request and the last incoming response. The varying parameter is the rate with which we send the requests. For this experiment, we use an Actinium app that simply calls `app.sleep(delay)` between sending the requests and measures the timing with `app.getNanoTime()`. The experiment is repeated 500 times, whereas we discard runs that do not complete within a maximum timeout of 20 s. This is caused by complete message loss in the LLN due to temporary interference. As a consequence, 1.8% of the 500 runs are discarded.

Figure 5.8 shows a drastic increase in latency at around 45 requests/s. This is where the LLN becomes congested and link-layer retransmissions exceed the channel capacity. Thus, an application-layer retransmission is required, which in a default CoAP configuration occurs after two to three seconds and is repeated after twice the previous interval until four retransmissions (see Chapter 2). Without interference, rates beyond the 45 requests/s mark can also achieve overall RTTs below 500 ms as indicated by the red minimum line.

5.3.5 Discussion

On the one hand, the Actinium evaluation shows that Rhino for Java 6 is not a high-performance runtime environment. On the other hand, arithmetic and logical operations perform comparatively well in JavaScript. This shows that scripting is well suited for the targeted use-case, where RESTful device resources are mashed up to create IoT applications. Memory-intensive tasks like persistent logging or data mining can be outsourced to stand-alone services with a RESTful API. At the time of the experiments, we were also bound to Rhino for the Java 6 JVM. With the new *InvokeDynamic* bytecode instruction in Java 7, the JVM provides better support for dynamically typed languages.[17] Furthermore, the recently released Java 8 provides a new JavaScript runtime implementation called *Nashorn*, which is more light-weight and performs better than Rhino [155]. This results in a speed-up for scripts, so that the JVM performs similar to the V8 runtime system.

[17] http://jcp.org/en/jsr/detail?id=292

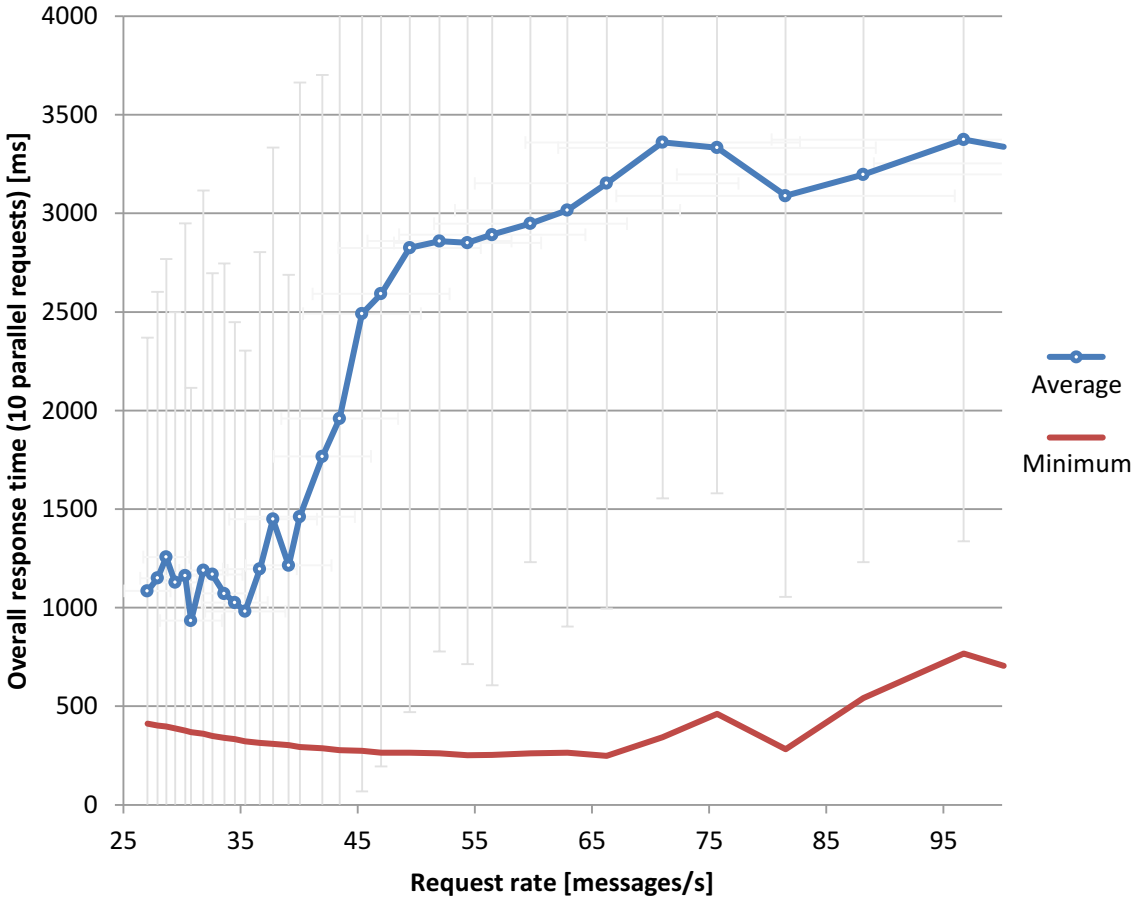


Figure 5.8: The overall response time of 10 responses for 10 requests sent with different rates. The graph also shows the measured minima, in which case no interference, thus no restransmissions, occurred. Note that the high average and standard deviation is caused by CoAP's binary exponential backoff for retransmissions, which starts with a random value between 2 and 3 seconds by protocol default.

To improve multitenancy for the Actinium runtime system, we used the results from the experiments to implement a simple rate limitation layer for Californium. This was mainly a case study for the advantages of scripting. The required benchmark was implemented in only a few lines of code and could be updated and deployed quickly. The achieved traffic shaping, however, only covers the scenario where the runtime system interacts with a single LLN with a known maximum rate. In a real-world deployment, IoT applications might use multiple LLNs with different channel properties. Thus, a proper solution should employ more dynamic mechanisms for rate limitation and make use of caching at the runtime container as well as the border of each LLN. In recent work, we replaced the simple rate limitation layer with an implementation of CoCoA [16, 22]. It allows for dynamic, per-endpoint traffic shaping based on real-time RTT measurements. Furthermore, Californium allows for caching based on the Guava library[18].

With sandboxing in Actinium and DTLS support through Scandium, we implement most of our security considerations. What needs to be addressed in future work is adequate, user-friendly tool support to sign app instances and provide proper resource authentication and authorization. This is connected to the ongoing work in the IETF where the ACE working group[19] is currently drafting a new authorization standard for the IoT. There must be a paradigm shift from perimeter security (i.e., all devices and applications in the local network are authorized) to individual authentication and authorization.

In conclusion, our open-source implementation shows that our concepts for a RESTful runtime container are feasible. The scripting model significantly eases development of IoT. Being based on well-known patterns from the Web, it also enables tech-savvy end-users to automate tasks. This can be improved further through graphical programming models that extend our concept as shown by Mainetti et al. [127].

[18] https://github.com/google/guava (accessed on 12 Feb 2015)
[19] http://tools.ietf.org/wg/ace/ (accessed on 12 Feb 2015)

5.4 Web Browser Support for the IoT

Developers and tech-savvy users require an appropriate tool for the IoT: to explore devices, test applications, and manage them both. Such a tool is particularly interesting for CoAP, which is designed for M2M communication and does not provide a standard presentation layer like HTML. Thus, we implemented an add-on for the *Mozilla Firefox* Web browser and thereby prototype the full Web experience for tiny IoT devices. Our *Copper (Cu) Coap user-agent* allows interaction with embedded Web resources by simply entering a CoAP URI into the address bar and using the RESTful methods GET, POST, PUT, and DELETE. It is comparable to other REST add-ons such as Poster[20] or RESTClient[21], but implements the Constrained Application Protocol (CoAP) and integrates it seamlessly into the Web browser. Users can browse devices, bookmark their resources like normal Web pages, and follow links in HTML documents to discover new devices. Our add-on can also render different Internet media types typically provided by devices.

5.4.1 User Interface Design

Copper primarily targets developers who want to explore, debug, and test RESTful Web services based on CoAP. For HTTP-based services, the Web browser is already a popular tool to do so. In addition to the basic GUI elements to browse resources and issue different requests, our add-on provides manual override for the full set of CoAP options and a detailed log, so developers can intensively test their applications or CoAP implementations. Our add-on can also render different Internet media types typically provided by devices, e.g., JSON as depicted in Figure 5.9. While tech-savvy users can also use Copper to configure their devices or retrieve data from them, it is not designed for inexperienced end-users of WoT applications. Our add-on is a generic browser for tiny resource-constrained IoT devices and thus is missing a presentation layer which is usually application-specific.

The add-on is called by simply entering a CoAP URI into the Web browser or following a link to a CoAP resource. At the top, it offers a toolbar for the RESTful methods, including a button for a direct observe request, and main tasks such as discovering all resources of a server or pinging the server to check availability. Furthermore, Copper offers a menu to configure its detailed behavior such as the request type (CON or NON), the preferred block size, and the observe cancellation method. Power-users can also activate a menu to directly execute the test requests defined in the latest ETSI Plugtest specification [64].

[20] https://addons.mozilla.org/firefox/addon/poster/ (accessed on 12 Feb 2015)
[21] https://addons.mozilla.org/firefox/addon/restclient/ (accessed on 12 Feb 2015)

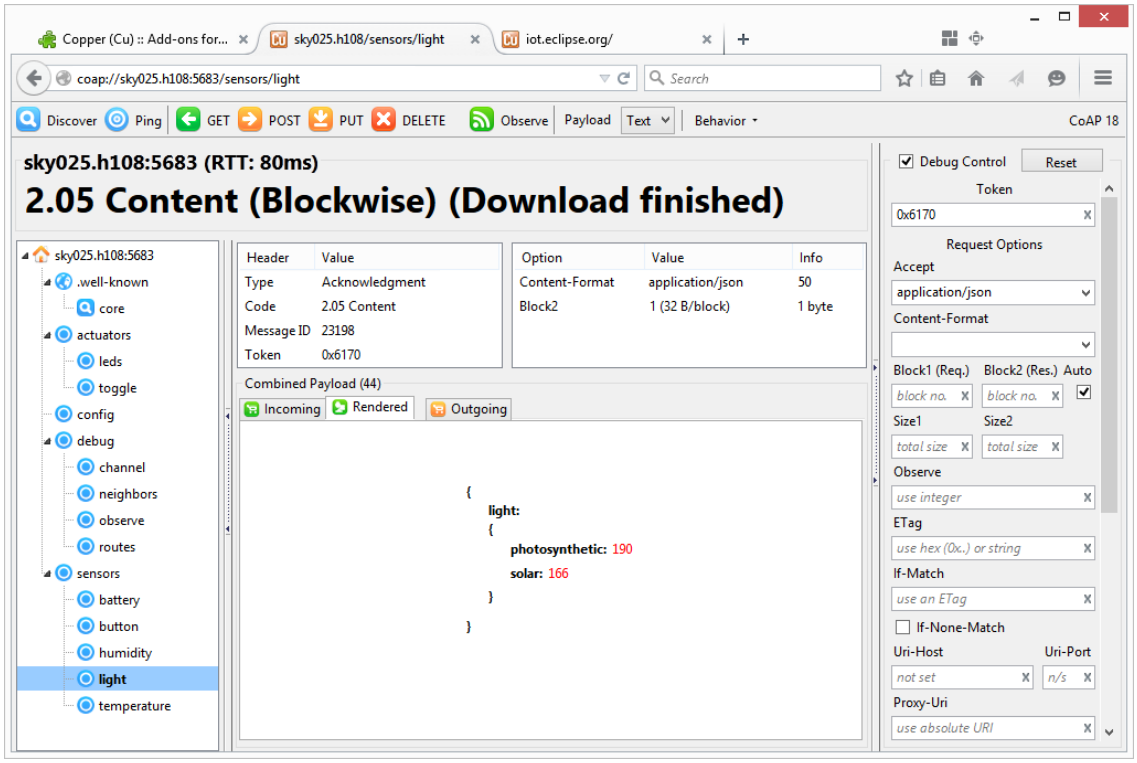


Figure 5.9: The server resources are discovered through the CoRE Link Format and shown on the left for browsing. The debug options on the right are optional and can be used to set header options to custom values for testing and debugging.

On the left, a tree view displays all resources that were discovered through the discovery resource (`/.well-known/core`) and the CoRE Link Format [168], similar to a file explorer. The icons depend on the Link Format attributes and indicate certain features such as observe support. The entries are used to navigate through the available Web resources They also support links to resources on other servers as found in the RD for instance.

The center shows the header and option information of the responses as well as the payload in the main view. For the latter, the user can choose between a raw dump and a rendered visualization such as the JSON tree shown in Figure 5.9. The large main view is also used to define the payload of outgoing requests. Alternatively, users can load the payload from files through the top toolbar.

Usually, the options of the outgoing request are set automatically by using the buttons and menus in the top toolbar. The panel on the right gives developers the possibility to individually define the options. This is particularly useful when debugging other CoAP implementations.

5.4.2 Copper (Cu) Implementation

Browser add-ons have more permissions than JavaScript code embedded on Web pages. The add-on code has access to the internal API of the browser and can even alter its appearance. Most Web browsers internally offer access to network sockets, including UDP, to implement custom protocols. Furthermore, add-ons can register a protocol handler for specific URI schemes. When a CoAP URI is entered, the browser loads the add-on and displays its GUI. In Mozilla products, the latter is defined in the XML User Interface Language (XUL), which extends XHTML with additional GUI elements the browser can render. The GUI layout is controlled by code like a normal Web page through loading scripts in the XUL document and defining callback functions, e.g., for pressing a button. The main script loads the JavaScript module that implements CoAP and opens a UDP socket to send messages to the addressed server.

Although Copper is fully written in JavaScript, for now CoAP requests cannot be issued from other scripts running in the browser. This means, JavaScript from an external Web page cannot include CoAP resources in an AJAX-like manner and user scripting is only available through editing the add-on sources, which are publicly available on GitHub[22]. Our assumption is that WoT mashups are mainly faceless scripts that augment and automate devices invisibly in the background, for which we propose the Actinium runtime container introduced above. The following study shows, however, that there is a broad interest in AJAX-like interaction with CoAP-enabled devices on Web pages to provide application-specific front-ends for end-users.

5.5 User Study and Trends

With Copper available since late 2010 and beyond 500 users according to the Mozilla add-on Web site[23], we were able to find 48 participants to conduct a user study on the WoT that, apart from researchers, also includes 16 industry developers. Our study focuses on the Web integration of resource-constrained devices in comparison to traditional networked embedded systems. The latter usually run proprietary protocols that are highly optimized for a given application, but increase development costs and cause technological silos. The study supports our hypothesis that Internet and Web protocols ease development in this domain. Furthermore, it also gives useful input on how to continue with the seamless integration of CoAP into the existing Web.

[22] https://github.com/mkovatsc/Copper (accessed on 12 Feb 2015)
[23] https://addons.mozilla.org/firefox/addon/copper-270430/ (accessed on 12 Feb 2015)

5.5.1 Hypotheses

Following up on the study by Guinard et al. [79], we published a questionnaire to evaluate the WoT vision in the context of highly resource-constrained networked embedded systems. In this domain, the use of IP and Web patterns is relatively new and numerous alternative, often proprietary, protocols exist. The goal of our study is to confirm or reject the following hypotheses:

1. Internet protocols and Web patterns ease the development of distributed software for resource-constrained devices.
2. CoAP is a required extension for the Web to integrate resource-constrained devices.
3. The Web browser is a preferred tool to interact with these devices when there is no physical interface meant for direct interaction.
4. Other tools are only preferred when a task is to be automated through a script or program.

5.5.2 Participants

To find enough experts who know and worked with both approaches and, hence, can give a qualified feedback, we advertised the study over the following channels:

1. The Contiki[24] and TinyOS[25] mailing lists, which reach mostly researchers in the area of Wireless Sensor Networks
2. ETSI M2M associates, who have a good overview of the available technologies for IoT and M2M solutions and mainly have an industry background
3. Followers of the IETF standardization, who often have many years of practical experience in the field

We received N=48 responses from people who worked with CoAP or IoT devices as researchers (51%), industry developers (34%), students (28%), hobbyists (9%), and lecturers (4%). Here, multiple roles are possible. The experience in the relevant fields (wireless sensor networks, traditional networked embedded devices, and Web technologies) varies from beginners to experts with up to 20 years of experience. The average experience with these technologies is 4.3 years among all participants.

[24] http://www.contiki-os.org/
[25] http://www.tinyos.net/

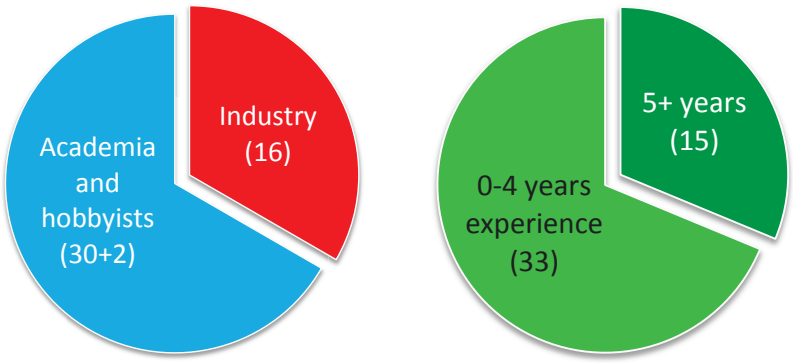


Figure 5.10: Our study involves 48 participants of which 16 are using IoT devices or CoAP as industry developers. Two participants are pure hobbyists. In the evaluation, we counted them as non-industry, i.e., academia.
The average experience is 4.3 years whereas 15 participants have been active in the field for at least five years.

5.5.3 IP, Web Patterns, and CoAP

For the first part, we used a five-level Likert scale from *strongly disagree* (0) to *strongly agree* (4). Hence, resulting values of more than 2 mean agreement with our statement. Based on the given background information (see Figure 5.10), we separated the responses into different groups: (i) academia, (ii) industry, (iii) less than five years of experience, and (iv) five or more years of experience. For each comparison among the groups (e.g., academia vs industry), we perform the *Wilcoxon rank-sum test* to see if the two sets significantly different from each other. For each statement, we give the corresponding p-value together with the sample sizes.

Figure 5.11 shows general agreement on our first hypothesis that Internet protocols and Web patterns ease the development. Overall, Internet protocols are slightly more accepted to ease development (3.3) than Web patterns (3.1). They are also more appreciated by participants with longer experience (3.7 vs 3.1 with less then five years).[26] Interestingly, participants with an industry background agree more with the advantages of Web patterns (3.5) than academia (3.0).[27]

Our second hypothesis about the necessity of CoAP is also confirmed, although the overall agreement is slightly lower (3.0) and has a slightly higher standard deviation (1.06). CoAP as additional Web protocol finds high acceptance among participants with an industry background (3.5 vs 2.8 in academia), though.[28]

[26] 0-4 vs 5+ years: p-value< 0.01, $R = 884.0$, $N_{0-4} = 33$, $N_{5+} = 15$
[27] Industry vs academia: p-value< 0.001, $R = 410.5$, $N_i = 16$, $N_a = 32$
[28] Industry vs academia: p-value< 0.001, $R = 441.5$, $N_i = 16$, $N_a = 32$

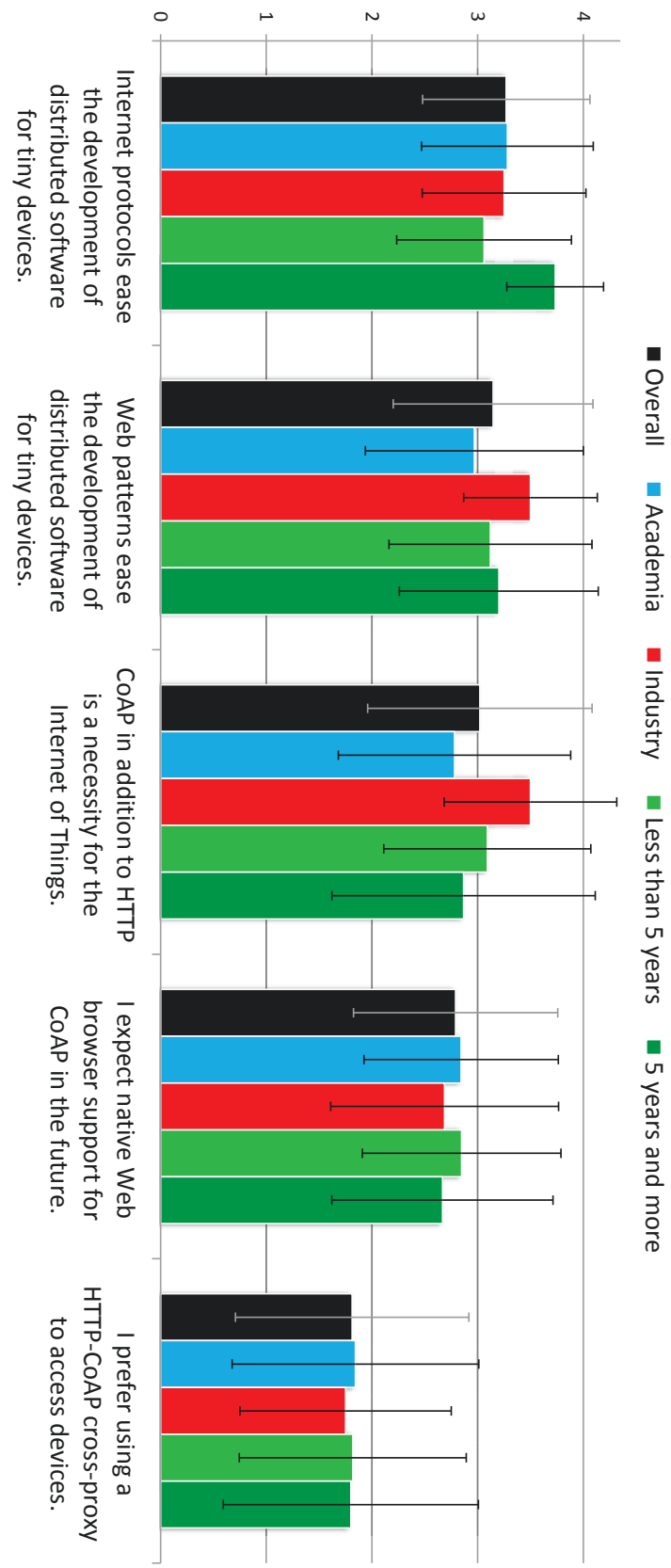


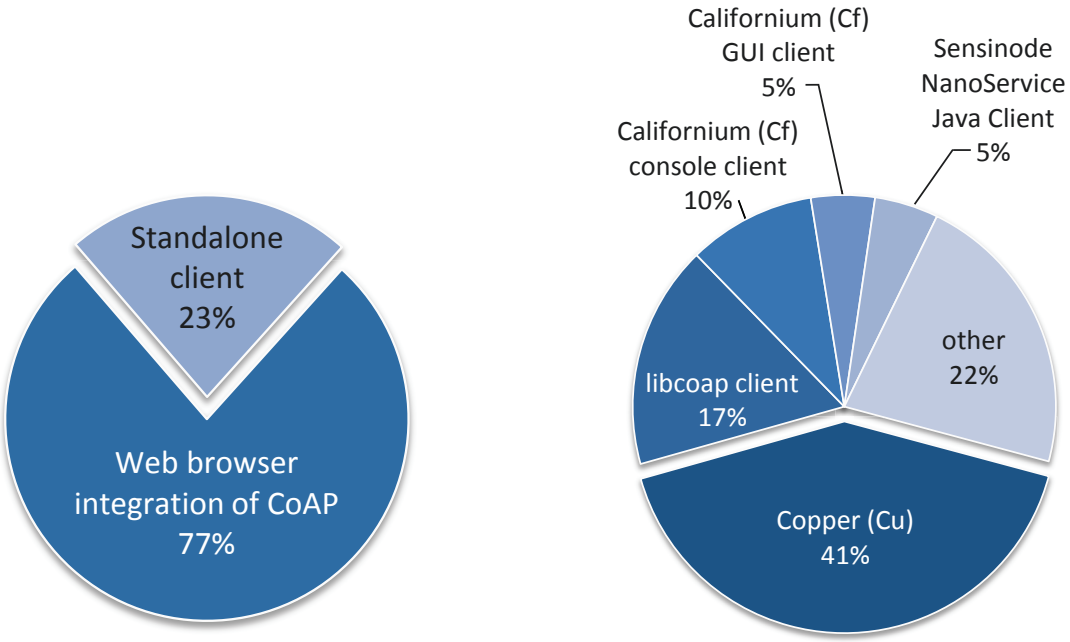
Figure 5.11: Likert scale responses by our 48 participants(0 = strongly disagree, 4 = strongly agree, error bars: ±1 std. dev.)

5.5.4 CoAP support in Web Browsers

Compared to the advantages of IP and Web patterns, the participants are slightly less confident about native CoAP support in Web browsers (2.8), although they rather disagree with the usage of an HTTP-CoAP cross-proxy (1.8). Conversely, the latter means that direct communication with devices is preferred.

In a second part of the questionnaire, we directly asked what way of user interaction with IoT devices is preferred. 77% voted for the Web browser over a standalone CoAP client. The most agreed-on reasons are that this way, no additional software is required (3.1, stdev. 0.9) and that the Web integration feels natural (3.0, stdev. 1.0). The only agreed-on reason for using a standalone CoAP client was that it is better suited for scripting and integration into a larger system (2.9, stdev. 0.7). Other reasons had more or less neutral outcome.

Figure 5.13 shows the client usage profiles of the participants. Being a contributor to the CoRE working group, our CoAP implementations have always been early reference implementations for others. Thus, one of the main use cases is debugging own implementations. The other applications are in line with the role of a Web browser in traditional RESTful Web services and reflect the preference for Web browser integration of CoAP.



(a) Preferred interaction model

(b) Client market shares

Figure 5.12: Most participants preferred the Web browser as client for device interaction. This is also reflected in the market share of different CoAP clients on workstations and laptops. Clients mentioned only once or without specific name are consolidated in "other."

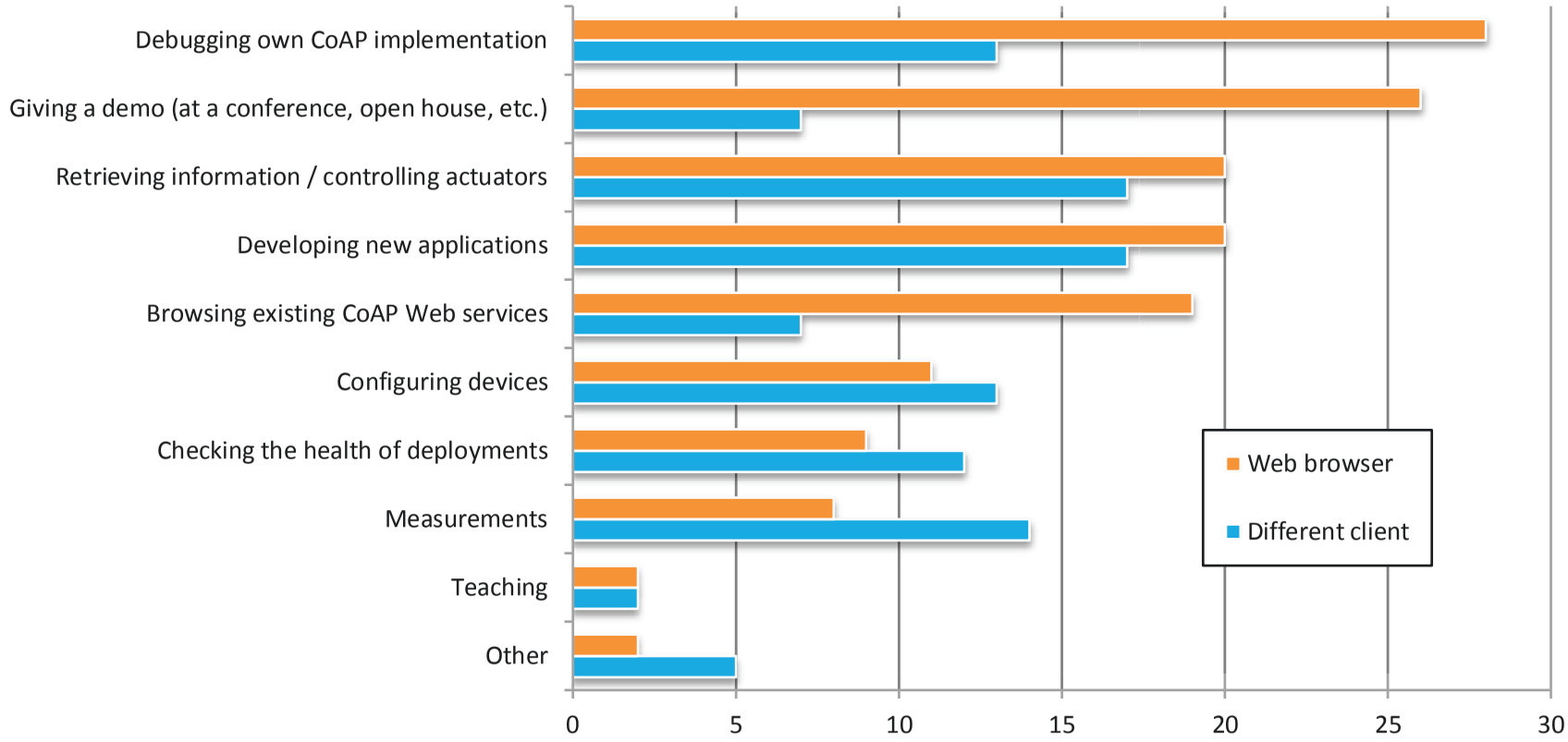


Figure 5.13: Copper has been available since CoAP draft 03. Thus, one of the main applications of it is the debugging of other CoAP implementations. Still, there have been other clients around and usability must have been the reason for choosing this reference implementation in particular.

5.5.5 CoAP Client Market Share

Copper is currently the only CoAP client available that integrates into the Web browser. Figure 5.12(b) shows that 41% of the participants who use CoAP on their workstation or laptop (N=41) use our browser add-on as primary client, followed by the command line *libcoap client* with 17%. The main reason for using a different CoAP client was by far the need for automation and scripting with 55% of the answers and the "I do not use Firefox" runner-up with mere 8%. Although the JavaScript source code of our add-on is available, only 2 participants (6%) have adapted it to their needs. 47% stated that they are "not familiar enough with Firefox add-ons" to do so, while the remaining 47% did not consider this option.

5.5.6 Discussion

Developers from industry and academia that deal with networked embedded systems are convinced that Internet protocols and patterns from the Web facilitate their job. Also the Web-like interaction with IoT devices is preferable, as 77% favor the Web browser integration of CoAP over standalone clients.

From the expectations but even more from the individual comments, we conclude that users prefer full CoAP support in Web browsers. A primary concern is the creation of intuitive Web front-ends that directly include device data without the need for a cross-proxy. We also take away that our CoapRequest object API is the right approach, since CoAP scripting support within the browser is highly appreciated by our participants. Scripting a Web site that is able to perform CoAP requests would, for instance, satisfy the feature requests for customized logging and visualization of historic values, since well-known libraries such as jQuery Flot[29] can easily be used for this.

5.6 Summary and Discussion

The goal of this thesis is to make the programming of IoT applications significantly easier by enabling Web technology for resource-constrained devices. To this end, we investigate new programming models and tools for networked embedded systems. Our thin server server architecture presented in Chapter 3 extends the WoT idea toward resource-constrained IoT devices. In this chapter, we presented our concept for a RESTful runtime container that addresses the key requirements for IoT applications. The script-based

[29] http://www.flotcharts.org/

IoT apps are modeled as resources themselves and can provide configuration parameters, status information, and results through a RESTful interface. Complex applications are implemented by mashing Web resources up, which can be provided directly by IoT devices, other apps, or classic Web services. To access the CoAP resources on the devices, we define the *CoapRequest* object API in compliance with the well-known XHR of AJAX. Security is provided through traditional Internet standards, but with a paradigm change in how applications are signed. We implemented our concept with the *Actinium (Ac) app-server for Californium* and evaluated its feasibility in a testbed deployment. Complementary to the faceless runtime container for automation tasks in the background, we prototype CoAP support in the Web browser, which allows direct user interaction with devices. Our *Copper (Cu) CoAP user-agent* is implemented as add-on for the Mozilla Firefox browser and is being used actively in development and teaching. It allows developers to test and debug CoAP-based services similar to classic Web services by interacting with the provided resources directly through the Web browser. To this end, we conducted a user study with 48 participants that confirms our hypotheses that Internet protocols and patterns from the Web ease the development of IoT applications.

Service frameworks such as Californium (see Chapter 4) are able to execute many apps in parallel to orchestrate a large number of devices. The evaluation of our runtime prototype shows that the performance requirements for a scripting language are relaxed by the latency of LLNs. Yet the JavaScript language performs comparably well for logic and arithmetic operations, which are dominant in IoT automation tasks. Thus, scripting and Web mashups are a viable programming model for the IoT given an adequate runtime system. Using a well-known scripting language also increases productivity and opens IoT application development to a broader audience. As witnessed in the World Wide Web, intuitive API and simple scripting language can even empower tech-savvy end-users to customize and create their own applications. To make IoT applications secure, runtime systems must provide adequate tool support to minimize human errors in the configuration of the security mechanisms. Our security model based on user-signed apps enables developers and end-users to build trusted IoT applications. An adoption of our *CoapRequest* object by Web browsers would further integrate resource-constrained IoT devices into the Web. An extension for CoAP, or more specifically for the DTLS-secured `coaps` scheme, would enable new scenarios that go beyond information on Web pages. Users could grant Web applications access to locally connected CoAP devices, similar to camera and microphone permissions, and benefit from better services. The manufacturer could use this for remote diagnostics or maintenance of its products. The Web application could also be used as a configuration tool for devices that do not provide any user interface or even for a complete home automation system. Due to the efforts of the IETF Real-Time Communication in WEB-browsers (RTCWEB) working group, the main browser vendors already started to integrate DTLS, e.g., for direct video chat or gaming [160].

Chapter 6

Conclusions

The goal of our thesis is to enable the ambitious vision of an Internet of Things (IoT) that connects hundreds of billions of devices. This requires seamless interoperability among devices and services at the application layer as well as improved usability over classic networked embedded systems. Both can be achieved by Web technology through the REST architectural style and the patterns well-known to developers and users. In this final chapter, we briefly revisit our findings and summarize our contributions along the three research questions of this dissertation: (i) How can we scale Web technology down to constrained environments? (ii) How can we scale Web technology up to hundreds of billions of IoT devices? (iii) How does Web technology improve usability for developers and users? We then conclude with an outlook on possible future work.

6.1 Summary

We started this dissertation by giving an overview over the Constrained Application Protocol (CoAP), which has been standardized within the Internet Engineering Task Force (IETF). This new Web protocol was designed from scratch following the REST architectural style. Unlike HTTP, it is tailored to the requirements of resource-constrained IoT devices and low-power IP networks. CoAP uses a compact binary format and runs over UDP. A messaging sub-layer adds a thin control layer that provides duplicate detection and reliable delivery of messages based on a simple stop-and-wait mechanism for retransmissions. On top, the request/response sub-layer enables RESTful interaction through the well-known methods GET, POST, PUT, and DELETE as well as response codes that are defined in close accordance to the HTTP specification. CoAP resources are addressable by URIs, and Internet Media Types are used to represent resource state. RESTful caching

and proxying enable network scalability. Yet CoAP offers features that go beyond HTTP, and hence make it a better fit for the IoT:

1. Resources are *observable*, that is, servers can push state changes to all registered clients through a request/multiple-response pattern.
2. This pattern also enables RESTful *group communication* where multiple servers respond to a request that is sent to an IP multicast address.
3. *Application-layer fragmentation* allows blockwise on-the-fly processing of messages that would otherwise exceed the maximum transmission unit (MTU) of 1280 bytes. This mechanism also helps with the potentially even smaller buffers of highly resource-constrained devices.
4. CoAP supports *alternative transports* such as SMS or TCP, while maintaining interoperability at the application layer.
5. Finally, CoAP includes an M2M *discovery mechanism* to find matching resources based on Web Linking. It uses either multicast or resource directories where devices can register on start-up.

In Chapter 3, we presented concepts and system architectures to scale Web technology down to tiny resource-constrained devices. Because new technologies will most likely be used to primarily minimize dimensions, power consumption, and unit costs, IoT devices will also remain resource-constrained. Our *thin server architecture* minimizes the system requirements for embedded Web servers by only providing the elementary hardware functions through a RESTful interface. The application logic is separated and moved outside the embedded domain, which lowers the entry barrier for IoT developers. This also allows for an application-agnostic infrastructure of IoT devices. It can serve multiple running applications at once, thereby enabling the convergence of application domains. We prototyped our thin server architecture based on CoAP and implemented the lightweight *Erbium (Er) REST Engine* for the Contiki operating system. In this course, we actively contributed to the protocol design in the CoRE working group at the IETF. Our concepts and experiences also helped to formulate guidelines for an efficient implementation of CoAP. Furthermore, we provided a comprehensive evaluation of CoAP in a realistic low-power setting. In low-power wireless systems, power-efficiency is determined by the ability to maintain a low radio duty cycle: keeping the radio off as much as possible. We leveraged the ContikiMAC duty cycling mechanism to provide power-efficiency for CoAP. We experimentally evaluated our low-power CoAP that leverages the ContikiMAC duty cycling mechanism. To the best of our knowledge, our CoAP implementation was the first to provide power-efficient operation through RDC. We showed that there is no need to optimize the theoretical number of link-layer frames and that link-layer bursts can significantly reduce the latency of requests. Our results also question the need for specialized low-power mechanisms at the application layer. Instead, low-power operation

can be added transparently through a separate RDC layer that provides virtual always-on semantics. All concepts are implemented in Erbium, which is open-source and became part of Contiki OS.

Chapter 4 tackled the question how to scale Web technology up to hundreds of billions of IoT devices. Emerging networking and backend support technology not only has to anticipate this dramatic increase in connected nodes, but also a change in traffic patterns. Instead of bulk data such as file sharing or multimedia streaming, IoT devices will primarily exchange real-time sensory and control data in small but numerous messages. Often cloud services will handle these data from a huge number of devices, and hence need to be extremely scalable to support conceivable large-scale IoT applications. To this end, we designed the CoAP-based *Californium system architecture* for IoT cloud services. It is inspired by state-of-the-art HTTP server architectures, which provide the basis for most classic cloud services. Our 3-stage architecture has a flexible concurrency model and can fully utilize modern multi-core platforms. It separates bookkeeping from message processing to achieve high throughput at vast concurrency factors To the best of our knowledge, we were the first to systematically evaluate the performance of CoAP in such unconstrained environments. Our Java-based *Californium (Cf) CoAP framework* outperforms other CoAP solutions and shows 33 to 64 times higher throughput than high-performance HTTP Web servers. The results substantiate that the low overhead of CoAP does not only enable Web technology for low-cost IoT devices, but also significantly improves backend service scalability for vast numbers of connected devices. The Californium implementation is open-source and publicly available at the Eclipse Foundation.

Chapter 5 was dedicated to the human in loop and we showed that Web technology improves the usability for IoT developers and users. Programming IoT applications is challenging because developers have to be knowledgeable in various technical domains, from low-power networking, over embedded operating systems, to distributed algorithms. Hence, it will be challenging to find enough experts to provide software for the vast number of expected devices. To help remedy this situation, we provided concepts and tools that enable Web-like scripting, debugging, and testing methods for the development of IoT applications. On the one hand, RESTful runtime containers form the counterpart for the thin server architecture. They expose scripts, their configurations, and their lifecycle management through a coherent RESTful interface. Such a runtime system can also provide user-friendly tool support for the security and privacy policies, which are fundamental in the IoT. Our *Actinium (Ac) app-server for Californium* is the implementation of such a runtime system. We endowed the JavaScript language with an API for direct interaction with thin servers, the *CoapRequest* object, and means to export script data as Web resources. With Actinium, IoT applications can be created by simply mashing up the elementary functions of devices, other scripts, and classic Web services. On the other

hand, Web browser integration of CoAP facilitates intuitive user interaction and brings the IoT closer to the World Wide Web. Our *Copper (Cu) CoAP user-agent* brings CoAP support to the Web browser and has been adopted by developers since late 2010. Thus, we were able to conduct a user study among industry and research developers who are familiar with both, our approach with CoAP and traditional programming models for networked embedded systems. The results showed general agreement with our hypothesis that IP and patterns from the Web ease development for IoT applications with tiny, resource-constrained devices. Actinium is part of the Californium project at the Eclipse Foundation. Copper is available as an add-on for the Firefox Web browser and open-source on GitHub.

6.2 Limitations and Future Work

We provide working open-source implementations for most of our concepts. They can be used as building blocks to create a working IoT system. However, there are still a number of limitations that pose interesting challenges for future work.

6.2.1 Resource-constrained Devices

Our Erbium (Er) REST Engine still needs to be integrated with an efficient DTLS implementation such as TinyDTLS. While this is mainly an engineering effort, it can be interesting to analyze the performance of hardware accelerators for cryptographic operations that are being integrated into IoT SoCs. A more fundamental issue is the lifecycle management of future IoT devices. In particular the (re-)commissioning phase with security bootstrapping is challenging: The devices usually have no direct user interfaces. While sensors can provide some form of input, actuators will only have the radio interface to receive input. Commissioning becomes even more challenging when assuming a home environment with consumer IoT devices. Finally, there must be a way to define privacy policies for resource-constrained devices. The missing user interface and their sheer number requires good tool support to make the policies manageable for end-users.

6.2.2 IoT Services

Our work and the standardization of CoAP only provides the architecture and protocol. Services in the IoT will require machine-readable descriptions that can be processed by resource-constrained devices. First, there must be meaningful Internet Media Types. Hypermedia does not only define the serialization format of representations, but also the

semantic processing model. The challenge is to define Internet Media Types that are both meaningful and re-usable in multiple application domains. Second, there should be semantic descriptions for automatic service composition or mashup creation. The Semantic Web already provides countless triples in open linked data to describe large parts of the world. However, the adoption in real-world systems is still low. Recently, bottom-up semantics shaped a more practical approach that focuses on contained environments that can be described with a limited number of triples. This appears promising for IoT services and applicable to resource-constrained devices.

6.2.3 Usability

Our user study showed a high interest in scriptable CoAP support in the Web browser. Having a powerful layout engine, Web browsers can provide graphical user interfaces (GUIs) for single IoT devices and whole applications that mash devices up with services from Web sites. Well-known technologies such as HTML, CSS, and JavaScript would significantly ease development and access from a wide range of user devices. There are still open issues, however, before browser vendors can integrate native CoAP support. Research must assess the threat model that results from the connection of arbitrary Web sites with potentially safety-critical IoT devices and find a suitable security model.

Bibliography

[1] IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Std 802.15.4-2003, 2003.

[2] SNAP based Wireless Mesh Networks. White paper, Synapse Wireless, June 2008.

[3] More Than 50 Billion Connected Devices. White Paper 284 23-3149 Uen, Ericsson, 2011.

[4] IEEE Draft Standard for Information Technology-Telecommunications and information exchange between systems-Local and Metropolitan networks-Specific requirements-Part II: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications-Amendment 8: IEEE 802.11 Wireless Network Management. IEEE P802.11v, Feb. 2012.

[5] In-transit Data Analysis and Distribution in a Multi-Cloud Environment using CometCloud. In *Proceedings of the International Workshop on Energy Management for Sustainable Internet-of-Things and Cloud Computing*, EMSICC, pages 1–6, Barcelona, Spain, 2014.

[6] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB, Seoul, Korea, 2006.

[7] D. Alessandrelli, M. Petracca, and P. Pagano. T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs. In *Proceedings of the 9th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 337–344, 2013.

[8] A. Azzarà, D. Alessandrelli, M. Petracca, and P. Pagano. Demonstration Abstract: PyoT, a Macroprogramming Framework for the IoT. In *Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, IPSN, pages 315–316, Berlin, Germany, 2014.

[9] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, Dec. 1997.

[10] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI, New Orleans, LA, USA, 1999.

[11] A. Banks and R. Gupta. MQTT Version 3.1.1. Candidate OASIS Standard 01, June 2014.

[12] P. Baronti, P. Pillai, V. W. Chook, S. Chessa, A. Gotta, and Y. F. Hu. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications*, 30(7):1655–1695, 2007. Wired/Wireless Internet Communications.

[13] M. Becker, K. Li, K. Kuladinithi, and T. Poetsch. Transport of CoAP over SMS. I-D: draft-becker-core-coap-sms-gprs-05, Aug. 2014.

[14] V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Evaluating the Scalability of Java Event-Driven Web Servers. In *Proceedings of the 33rd International Conference on Parallel Processing*, ICPP, pages 134–142, Montreal, Canada, 2004.

[15] V. Beltran, J. Torres, and E. Ayguade. Understanding Tuning Complexity in Multithreaded and Hybrid Web Servers. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–12, Miami, FL, USA, 2008.

[16] A. Betzler, C. Gomez, I. Demirkol, and M. Kovatsch. Congestion Control for CoAP Cloud Services. In *Proceedings of the 8th International Workshop on Service-Oriented Cyber-Physical Systems in Converging Networked Environments*, SOCNE, pages 7–12, Barcelona, Spain, 2014.

[17] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells. Congestion control in reliable coap communication. In *Proceedings of the 16th ACM International Conference on Modeling, Analysis & Simulation of Wireless and Mobile Systems*, MSWiM, pages 365–372, Barcelona, Spain, 2013.

[18] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A Scientific Instrument for Precision Sensing and Data Recovery in Environmental Extremes. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, pages 265–276, San Francisco, CA, USA, 2009.

[19] M. Blackstock and R. Lea. IoT Mashups with the WoTKit. In *Proceedings of the 3rd International Conference on the Internet of Things*, IoT, pages 159–166, Wuxi, China, 2012.

[20] M. S. Blumenthal and D. D. Clark. Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World. *Transactions on Internet Technology*, 1(1):70–109, Aug. 2001.

[21] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC, pages 13–16, Helsinki, Finland, 2012.

[22] C. Bormann, A. Betzler, C. Gomez, and I. Demirkol. CoAP Simple Congestion Control/Advanced. I-D: draft-bormann-core-cocoa-02, July 2014.

[23] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228 (Informational), May 2014.

[24] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), Oct. 2013.

[25] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. I-D: draft-ietf-core-block-16, Oct. 2014.

[26] A. Boulis, C. Han, and M. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 187–200, San Francisco, CA, USA, 2003.

[27] M. Boussard, B. Christophe, O. Le Berre, and V. Toubiana. Providing user support in Web-of-Things enabled Smart Spaces. In *Proceedings of the 2nd International Workshop on the Web of Things*, WoT, pages 1–6, San Francisco, CA, USA, 2011.

[28] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest Information Discovery and Access System. *Computer Networks and ISDN Systems*, 28(1-2):119–125, Dec. 1995.

[29] A. Brandt and J. Buron. Transmission of IPv6 Packets over ITU-T G.9959 Networks. RFC 7428 (Proposed Standard), Feb. 2015.

[30] D. L. Brock. The Electronic Product Code (EPC) – A Naming Scheme for Physical Objects. White Paper MIT-AUTOID-WH-002, MIT Auto-ID Center, Jan. 2001.

[31] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, A Feature-Rich VM for the Resource Poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 169–182, Berkeley, CA, USA, 2009.

[32] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys, pages 307–320, 2006.

[33] T. Butt, I. Phillips, L. Guan, and G. Oikonomou. Adaptive and context-aware service discovery for the internet of things. In S. Balandin, S. Andreev, and Y. Koucheryavy, editors, *Internet of Things, Smart Spaces, and Next Generation Networking*, volume 8121 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 2013.

[34] D. Carrera, V. Beltran, J. Torres, and E. Ayguade. A Hybrid Web Server Architecture for E-Commerce Applications. In *Proceedings of the 11th IEEE International Conference on Parallel and Distributed Systems*, ICPADS, pages 182–188, Fuduoka, Japan, 2005.

[35] A. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web Services for the Internet of Things through CoAP and EXI. In *Proceedings of the 2011 IEEE International Conference on Communications*, ICC, pages 1–6, Kyoto, Japan, 2011.

[36] A. Castellani, S. Loreto, A. Rahman, T. Fossati, and E. Dijk. Guidelines for HTTP-CoAP Mapping Implementations. I-D: draft-ietf-core-http-mapping-05, Oct. 2014.

[37] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das. A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines. In *Proceedings of the 14th International World Wide Web Conference*, WWW, pages 730–739, Chiba, Japan, 2005.

[38] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari. IoT-OAS: An OAuth-based Authorization Service Architecture for Secure Services in IoT Scenarios. *Sensors Journal*, PP(99):1–11, 2014.

[39] S. Cirani, M. Picone, and L. Veltri. mjCoAP: An Open-Source Lightweight Java CoAP Library for Internet of Things Applications. In *22nd International Conference on Software, Telecommunications and Computer Networks*, SoftCOM, Split, Croatia, 2014.

[40] W. Colitti, K. Steenhaut, and N. De Caro. Integrating Wireless Sensor Networks with the Web. In *Proceedings of the 2011 Workshop on Extending the Internet to Low power and Lossy Networks*, IP+SN, pages 1–5, Chicago, IL, USA, 2011.

[41] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. sMAP: A Simple Measurement and Actuation Profile for Physical Information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys, Zurich, Switzerland, 2010.

[42] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[43] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proceedings of the 1st IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS, pages 340–348, Fort Lauderdale, FL, USA, 2004.

[44] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 85–98, San Francisco, CA, USA, 2003.

[45] A. Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011:13, SICS Swedish ICT, 2011.

[46] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes. Powertrace: Network-Level Power Profiling for Low-power Wireless Networks. Technical Report T2011:05, SICS Swedish ICT, 2011.

[47] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys, pages 15–28, Boulder, CO, USA, 2006.

[48] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th IEEE Conference on Local Computer Networks*, LCN, pages 455–462, Tampa, FL, USA, 2004.

[49] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, and N. Finne. The Announcement Layer: Beacon Coordination for the Sensornet Stack. In *Proc. EWSN*, Bonn, Germany, 2011.

[50] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys, pages 29–42, Boulder, CO, USA, 2006.

[51] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Smews: Smart and Mobile Embedded Web Server. In *Proceedings of the 2009 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS, pages 571–576, Fukuoka, Japan, 2009.

[52] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The Web of Things: Interconnecting Devices with High Usability and Performance. In *Proceedings of the 6th International Conference on Embedded Software and Systems*, ICESS, pages 323–330, Hangzhou, China, 2009.

[53] S. Duquennoy, O. Landsiedel, and T. Voigt. Let the Tree Bloom: Scalable Opportunistic Routing with ORPL. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 1–14, Roma, Italy, 2013.

[54] S. Duquennoy, N. Wirström, and A. Dunkels. Demo: Snap: Rapid sensornet deployment with a sensornet appstore. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 405–406, Seattle, WA, USA, 2011.

[55] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making Sensor Networks IPv6 Ready. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys, pages 421–422, Raleigh, NC, USA, 2008.

[56] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and Evaluation of a Versatile and Efficient Receiver-initiated Link Layer for Low-power Wireless. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 1–14, 2010.

[57] ECMA Int. ECMAScript Language Specification 5.1. ECMA-262, 2011.

[58] K. L. (Ed.), J. Martocci, C. Neilson, and S. Donaldson. Transmission of IPv6 over MS/TP Networks. I-D: draft-ietf-6lo-6lobac-00, July 2014.

[59] P. M. (Ed.), J. Petersen, Z. Shelby, M. V. de Logt, and D. Barthel. Transmission of IPv6 Packets over DECT Ultra Low Energy. I-D: draft-ietf-6lo-dect-ule-01, Jan. 2015.

[60] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, ACM MobiCom, pages 263–270, Seattle, WA, USA, 1999.

[61] ETSI. 1st CoAP Plugtest; Paris, France; 24 – 25 March 2012. Technical Report CTI Plugtest Report 1.1.1 (2012-03), ETSI, 2012.

[62] ETSI. CoAP#2 Plugtest; Sophia-Antipolis, France; 28 – 30 November 2012. Technical Report CTI Plugtest Report 1.0.0 (2013-01), ETSI, 2013.

[63] ETSI. CoAP#3 and OMA LWM2M Plugtests; Las Vegas, USA; 19 – 22 November 2013. Technical Report CTI Plugtest Report 1.0.0 (2013-12), ETSI, 2013.

[64] ETSI. CoAP#4 Plugtests; London, UK; 7 – 9 March 2014. Technical Report CTI Plugtest Report 1.0.0 (2014-03), ETSI, 2014.

[65] O. Evangelatos, K. Samarasinghe, and J. Rolim. Syndesi: A Framework for Creating Personalized Smart Environments Using Wireless Sensor Networks. In *Proceedings of the 9th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 325–330, Cambridge, MA, USA, 2013.

[66] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 4601 (Proposed Standard), Aug. 2006. Updated by RFCs 5059, 5796, 6226.

[67] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power Wireless Bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys, pages 1–14, Toronto, Canada, 2012.

[68] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011.

[69] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[70] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard), June 2014.

[71] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[72] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[73] H. Gellersen. Smart-its: Computers for artifacts in the physical world. *Communications of the ACM*, 48(3):66, Mar. 2005.

[74] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 1–14, Berkeley, CA, USA, 2009.

[75] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The Tenet Architecture for Tiered Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys, pages 153–166, Boulder, Colorado, USA, 2006.

[76] D. Gordon, M. A. Neumann, and M. Beigl. Program Your Reality with dinammite. In *Demos at the 9th International Conference on Pervasive Computing*, PERVASIVE, San Francisco, CA, USA, 2011.

[77] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI Template. RFC 6570 (Proposed Standard), Mar. 2012.

[78] S. Guha and P. Francis. Characterization and Measurement of TCP Traversal Through NATs and Firewalls. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC, pages 199–211, Berkeley, CA, USA, 2005.

[79] D. Guinard, I. Ion, and S. Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In *Proceedings of the 8th International ICST Conference on Mobile and Ubiquitous Systems*, Mobiquitous, pages 326–337, Copenhagen, Denmark, 2011.

[80] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards Physical Mashups in the Web of Things. In *Proceedings of the 6th International Conference on Networked Sensing Systems*, INSS, pages 1–4, Pittsburgh, PA, USA, 2009.

[81] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of the 2nd International Conference on the Internet of Things*, IoT, pages 1–8, Tokyo, Japan, 2010.

[82] V. Gupta, J. Kim, A. Pandya, K. Lakshmanan, R. Rajkumar, and E. Tovar. Nano-CF: A Coordination Framework for Macro-Programming in Wireless Sensor Networks.

In *Proceedings of the 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON, pages 467–475, Salt Lake City, UT, USA, 2011.

[83] A. S. Harji, P. A. Buhr, and T. Brecht. Comparing High-Performance Multi-Core Web-Server Architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR, pages 1–12, Haifa, Israel, 2012.

[84] K. Hartke. Observing Resources in CoAP. I-D: draft-ietf-core-observe-16, Dec. 2014.

[85] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, HICSS, pages 1–10, Maui, HI, USA, 2000.

[86] K. Hewage and T. Voigt. Towards TCP Communication with the Low Power Wireless Bus. In *The 11th ACM Conference on Embedded Network Sensor Systems*, SenSys, pages 1–2, Roma, Italy, 2013.

[87] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. MacroLab: A Vector-based Macroprogramming Framework for Cyber-physical Systems. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 225–238, Raleigh, NC, USA, 2008.

[88] J. Hoebeke, D. Carels, I. Ishaq, G. Ketema, J. Rossey, E. Depoorter, I. Moerman, and P. Demeester. Leveraging Upon Standards to Build the Internet of Things. In *Proceedings of the 19th IEEE Symposium on Communications and Vehicular Technology in the Benelux*, SCVT, pages 1–6, 2012.

[89] G. Holland and N. Vaidya. Analysis of TCP Performance over Mobile Ad Hoc Networks. *Wireless Networks*, 8(2/3):275–288, Mar. 2002.

[90] J. Hui and D. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 15–28, Raleigh, NC, USA, 2008.

[91] J. Hui and R. Kelsey. Multicast Protocol for Low power and Lossy Networks (MPL). draft-ietf-roll-trickle-mcast-09, Apr. 2014.

[92] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), Sept. 2011.

[93] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol

for Network Programming at Scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys, pages 81–94, Baltimore, MD, USA, 2004.

[94] ISO/IEC. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models. ISO 25010, Mar. 2011.

[95] C. Jennings, Z. Shelby, and J. Arkko. Media Types for Sensor Markup Language (SENML). I-D: draft-jennings-core-senml-00, Nov. 2014.

[96] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and Implementation of a High-Fidelity AC Metering Network. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, pages 253–264, San Francisco, CA, USA, 2009.

[97] X. Jiang, M. Van Ly, J. Taneja, P. Dutta, and D. Culler. Experiences with a high-fidelity wireless building energy auditing network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys, pages 113–126, Berkeley, CA, USA, 2009.

[98] Julian Aubourg and Jungkee Song and Hallvord R. M. Steen. XMLHttpRequest. Wide Web Consortium WD-XMLHttpRequest-20121206.

[99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next Century Challenges: Mobile Networking for Smart Dust. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, ACM MobiCom, pages 271–278, Seattle, WA, USA, 1999.

[100] Kamiya, T. and J. Schneider. Efficient XML Interchange (EXI) Format 1.0. World Wide Web Consortium REC-exi-20110310.

[101] A. Kansal, S. Nath, J. Liu, and F. Zhao. SenseWeb: An Infrastructure for Shared Sensing. *MultiMedia*, 14(4):8–13, 2007.

[102] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *SIGCOMM Computer Communication Review*, 17(5):2–7, 1987.

[103] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Dec. 2005.

[104] M. Kovatsch. Firm Firmware and Apps for the Internet of Things. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*,

SESENA, pages 61–62, Honolulu, HI, USA, 2011.

[105] M. Kovatsch. CoAP for the Web of Things: From Tiny Resource-constrained Devices to the Web Browser. In *Proceedings of the 4th International Workshop on the Web of Things*, WoT, pages 1–9, Zurich, Switzerland, 2013.

[106] M. Kovatsch, O. Bergmann, E. Dijk, X. He, and C. Bormann. CoAP Implementation Guidance. I-D: draft-ietf-lwig-coap-01, July 2014.

[107] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the 8th IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS, pages 855–860, Valencia, Spain, 2011.

[108] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In *Proceedings of the 3rd International Conference on the Internet of Things*, IoT, pages 135–142, Wuxi, China, 2012.

[109] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable Cloud Services for the Internet of Things with CoAP. In *Proceedings of the 4th International Conference on the Internet of Things*, IoT, pages 135–142, Cambridge, MA, USA, 2014.

[110] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS, pages 751–756, Palermo, Italy, 2012.

[111] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proceedings of the 2011 Workshop on Extending the Internet to Low power and Lossy Networks*, IP+SN, pages 1–7, Chicago, IL, USA, 2011.

[112] Z. T. Kyaw. NanoService™Whitepaper. White paper, Sensinode, Oct. 2012.

[113] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, IPDPS, pages 1–8, Rhodes Island, Greece, 2006.

[114] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and*

Tools for Embedded Systems, LCTES, pages 182–187, Snow Bird, UT, USA, 2001.

[115] S. Lemay, V. S. Barboza, and H. Tschofenig. A TCP and TLS Transport for the Constrained Application Protocol (CoAP). I-D: draft-tschofenig-core-coap-tcp-tls-01, Sept. 2014.

[116] C. Lerche, N. Laum, F. Golatowski, D. Timmermann, and C. Niedermeier. Connecting the Web with the Web of Things: Lessons Learned from Implementing a CoAP-HTTP Proxy. In *Supplement Volume / Workshops of the 9th IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS, pages 1–8, Las Vegas, NV, USA, 2012.

[117] C. Lerche, N. Laum, G. Moritz, E. Zeeb, F. Golatowski, and D. Timmermann. Implementing Powerful Web Services for Highly Resource-Constrained Devices. In *Workshop Proceedings of the 9th Annual IEEE International Conference on Pervasive Computing and Communications*, PERCOM Workshops, pages 332–335, Seattle, WA, USA, 2011.

[118] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 85–95, San Jose, CA, USA, 2002.

[119] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[120] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design & Implementation*, NSDI, pages 15–28, San Francisco, CA, USA, 2004.

[121] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202 (Informational), Apr. 2011.

[122] O. M. A. Ltd. Lightweight Machine to Machine Technical Specification. OMA-TS-LightweightM2M-V1 0-20131210-C, Dec. 2013.

[123] A. Ludovici, P. Moreno, and A. Calveras. TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS. *Journal of Sensor and Actuator*

Networks, 2(2):288–315, 2013.

[124] C. MacGillivray, V. Turner, and D. Lund. Gartner Says It's the Beginning of a New Era: The Digital Industrial Economy. Online http://www.gartner.com/newsroom/id/2602817, Gartner Inc., 2013.

[125] C. MacGillivray, V. Turner, and D. Lund. Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars. Market Analysis 243661, IDC, 2013.

[126] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *Transactions on Database Systems*, 30(1):122–173, 2005.

[127] L. Mainetti, V. Mighali, L. Patrono, P. Rametta, and S. Oliva. A novel architecture enabling the visual implementation of web of things applications. In *Proceedings of he 21st International Conference on Software, Telecommunications and Computer Networks*, SoftCOM, pages 1–7, 2013.

[128] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA, pages 88–97, Atlanta, GA, USA, 2002.

[129] F. Mattern. *Die technische Basis für M2M und das Internet der Dinge*, pages 46–69. Münchner Kreis, 2013.

[130] F. Mattern and C. Floerkemeier. *From the Internet of Computers to the Internet of Things*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010.

[131] S. Mayer, A. Tschofen, A. K. Dey, and F. Mattern. User Interfaces for Smart Things - A Generative Approach with Semantic Interaction Descriptions. *Transactions on Computer-Human Interaction*, 21(2), 2014.

[132] O. Mazhelis, M. Waldburger, G. Machado, B. Stiller, and P. Tyrväinen. Retrieving Monitoring and Accounting Information from Constrained Devices in Internet-of-Things Applications. In *Proceedings of the 7th IFIP International Conference on Autonomous Infrastructure, Management, and Security – Emerging Management Mechanisms for the Future Internet*, AIMS, pages 136–147. Barcelona, Spain, 2013.

[133] B. Metcalfe. Metcalfe's Law after 40 Years of Ethernet. *Computer*, 46(12):26–31, Dec. 2013.

[134] A. Monacchi, D. Egarter, and W. Elmenreich. Integrating Households into the Smart Grid. In *Proceedings of the 2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems*, MSCPES, pages 1–6, Berkeley, CA, USA, 2013.

[135] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), Sept. 2007. Updated by RFCs 6282, 6775.

[136] G. Moritz, F. Golatowski, and D. Timmermann. A Lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks. In *Proceedings of the 8th IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS, pages 861–866, Valencia, Spain, 2011.

[137] G. Moritz, F. Golatowski, and D. Timmermann. A Lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks. In *Proceedings of the 8th IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS, pages 861–866, Valencia, Spain, 2011.

[138] S. Murugesan. Understanding Web 2.0. *IT Professional*, 9(4):34–41, July 2007.

[139] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby, and C. Gomez. Transmission of IPv6 Packets over BLUETOOTH(R) Low Energy. I-D: draft-ietf-6lo-btle-07, Jan. 2015.

[140] M. Nottingham. Web Linking. RFC 5988 (Proposed Standard), Oct. 2010.

[141] M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785 (Proposed Standard), Apr. 2010.

[142] D. I. Oberstag and D. Pauli. *Californium: A CoAP Framework in Java*. Lab project thesis, Department of Computer Science, ETH Zurich, 2011.

[143] B. Ostermaier, M. Kovatsch, and S. Santini. Connecting Things to the Web using Programmable Low-power WiFi Modules. In *Proceedings of the 2nd International Workshop on the Web of Things*, WoT, pages 1–6, San Francisco, CA, USA, 2011.

[144] B. Ostermaier, F. Schlup, and K. Römer. WebPlug: A Framework for the Web of Things. In *Workshop Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications*, PerCom, pages 690–695, Mannheim, Germany, 2010.

[145] J. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

[146] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: an Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference*, USENIX, pages 1–14, Monterey, CA, USA, 1999.

[147] S. Palchaudhuri, R. Kumar, and A. K. Saha. A Web Server Architecture for Symmetric Multiprocessor System. Project Report for Comp520, Department of Computer Science, Rice University, 2000.

[148] R. K. Panta, S. Bagchi, and S. P. Midkiff. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX, pages 1–14, San Diego, CA, USA, 2009.

[149] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the Performance of Web Server Architectures. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2007)*, pages 231–243, Lisbon, Portugal, 2007.

[150] A. Paventhan, S. Krishna, H. Krishna, R. Kesavan, and N. Ram. WSN Monitoring for Agriculture: Comparing SNMP and Emerging CoAP Approaches. In *Proceedings of the India Educators' Conference*, TIIEC, pages 353–358, Bangalore, India, 2013.

[151] C. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298, 2011.

[152] J. L. Pérez, Á. Villalba, D. Carrera, I. Larizgoitia, and V. Trifa. The COMPOSE API for the Internet of Things. In *Companion Volume of the 23rd International World Wide Web Conference*, WWW, pages 971–976, Seoul, Korea, 2014.

[153] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys, pages 95–107, Baltimore, MD, USA, 2004.

[154] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN, pages 364–369, Apr. 2005.

[155] J. Ponge. Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM. *Oracle Java Magazine*, January/February, 2014.

[156] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[157] G. J. Pottie and W. J. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[158] A. Rahman and E. Dijk. Group Communication for the Constrained Application Protocol (CoAP). RFC 7390 (Experimental), Oct. 2014.

[159] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. Lithe: Lightweight Secure CoAP for the Internet of Things. *Sensors Journal*, 13(10):3711–3720, Oct. 2013.

[160] E. Rescorla. WebRTC Security Architecture. I-D: draft-ietf-rtcweb-security-arch-10, July 2014.

[161] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012.

[162] K. Römer and F. Mattern. The Design Space of Wireless Sensor Networks. *Wireless Communications*, 11(6):54–61, Dec. 2004.

[163] C. Roduner and M. Langheinrich. BIT – A Framework and Architecture for Providing Digital Services for Physical Products. In *Proceedings of the 2nd International Conference on the Internet of Things*, IoT, pages 1–8, Tokyo, Japan, 2010.

[164] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi. SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes. *Transactions on Mobile Computing*, 9(12):1749–1765, 2010.

[165] S. Sarma, D. L. Brock, and K. Ashton. The Networked Physical World – Proposals for Engineering the Next Generation of Computing, Commerce & Automatic-Identification. White Paper MIT-AUTOID-WH-001, MIT Auto-ID Center, Oct. 2000.

[166] D. C. Schmidt. Using Design Patterns to Develop Reusable Object-oriented Communication Software. *Communications of the ACM*, 38(10):65–74, 1995.

[167] L. Schor, P. Sommer, and R. Wattenhofer. Towards a Zero-Configuration Wireless Sensor Network Architecture for Smart Buildings. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys, pages 31–36, Berkeley, CA, USA, 2009.

[168] Z. Shelby. Constrained RESTful Environments (CoRE) Link Format. RFC 6690 (Proposed Standard), Aug. 2012.

[169] Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet (Wiley Series on Communications Networking & Distributed Systems)*. Wiley, 2010.

[170] Z. Shelby and C. Bormann. CoRE Resource Directory. I-D: draft-ietf-core-resource-directory-02, Nov. 2014.

[171] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775 (Proposed Standard), Nov. 2012.

[172] Z. Shelby, K. Hartke, and C. Bormann. Constrained Application Protocol (CoAP). RFC 7252, 2013.

[173] Z. Shelby, P. Mahonen, J. Riihijarvi, O. Raivio, and P. Huuskonen. NanoIP: The Zen of Embedded Networking. In *Proceedings of the 2003 IEEE International Conference on Communications*, ICC, pages 1218–1222, Anchorage, AK, USA, 2003.

[174] Z. Shelby and M. Vial. CoRE Interfaces. I-D: draft-ietf-core-interfaces-02, Nov. 2014.

[175] B. Silverajan and T. Savolainen. CoAP Communication with Alternative Transports. I-D: draft-silverajan-core-coap-alternative-transports-07, Dec. 2014.

[176] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, and M. Nixon. WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 377–386, St. Louis, MO, USA, 2008.

[177] V. Tanyingyong, R. Olsson, M. Hidell, P. Sjodin, and B. Pehrson. Design and implementation of an IoT-controlled DC-DC converter. In *Proceedings of the 3rd IFIP Conference on Sustainable Internet and ICT for Sustainability*, SustainIT, pages 1–4, 2013.

[178] G. Teklemariam, J. Hoebeke, I. Moerman, and P. Demeester. Facilitating the creation of IoT applications through conditional observations in CoAP. *EURASIP Journal on Wireless Communications and Networking*, 2013(1):177, 2013.

[179] G. K. Teklemariam, J. Hoebeke, F. Van den Abeele, I. Moerman, and P. Demeester. Simple RESTful sensor application development model using CoAP. In *Workshop Proceedings of the 39th IEEE Conference on Local Computer Networks*, LCN Workshops, pages 552–556, Edmonton, Canada, 2014.

[180] G. Tolle. Embedded Binary HTTP (EBHTTP). I-D: draft-tolle-core-ebhttp-00, Mar. 2010.

[181] D. Tracey and C. Sreenan. A Holistic Architecture for the Internet of Things, Sensing Services and Big Data. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pages 546–553, Delft, The Netherlands, 2013.

[182] V. Trifa, S. Wieland, D. Guinard, and T. M. Bohnert. Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices. In *Proceedings of the 2nd International Workshop on Sensor Network Engineering*, IWSNE, pages 1–14, Marina del Rey, CA, USA, 2009.

[183] N. Tsiftes, J. Eriksson, and A. Dunkels. Low-power Wireless IPv6 Routing with ContikiRPL. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, pages 406–407, Stockholm, Sweden, 2010.

[184] Y.-H. Tu, Y.-C. Li, T.-C. Chien, and P. H. Chou. EcoCast: Interactive, Object-Oriented Macroprogramming for Networks of Ultra-Compact Wireless Sensor Nodes. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, pages 366–377, Chicago, IL, USA, 2011.

[185] L. M. Vaquero and L. Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *SIGCOMM Computer Communication Review*, 44(5):27–32, Oct. 2014.

[186] J.-P. Vasseur and A. Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann, 2010.

[187] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, and K. Pister. A Realistic Energy Consumption Model for TSCH Networks. *Sensors Journal*, 14(2):482–489, Feb. 2014.

[188] R. von Behren, J. Condit, and E. Brewer. Why Events are a Bad Idea (for High-Concurrency Servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, HotOS, pages 19–24, Lihue, Hawaii, 2003.

[189] W3C. XMLHttpRequest Level 2. W3C Working Draft 17 Jan 2012.

[190] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister. OpenWSN: A Standards-based Low-power Wireless Development En-

vironment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.

[191] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991.

[192] M. Welsh, D. Culler, and E. Brewer. SEDA: an Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP, pages 230–243, Banff, Canada, 2001.

[193] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design & Implementation*, NSDI, pages 29–42, San Francisco, CA, USA, 2004.

[194] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring Volcanic Eruptions with a Wireless Sensor Network. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks*, EWSN, pages 108–120, Istanbul, Turkey, 2005.

[195] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. Deploying a Wireless Sensor Network on an Active Volcano. *Internet Computing*, 10(2):18–25, Mar. 2006.

[196] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *Proceedings of the 5th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, pages 416–423, Nashville, TN, USA, 2006.

[197] E. Wilde. Putting Things to REST. Technical Report 2007-015, School of Information, UC Berkeley, Berkeley, CA, USA, 2007.

[198] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), Mar. 2012.

[199] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, 2014.

[200] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson. M2M: From Mobile to Embedded Internet. *Communications Magazine*, 49(4):36–43, 2011.

[201] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record*, 31(3):9–18, 2002.

[202] D. Yazar and A. Dunkels. Efficient Application Integration in IP-Based Sensor Networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys, pages 43–48, Berkeley, CA, USA, 2009.

[203] W. Ye, J. Heidemann, and D. Estrin. An Energy-efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM, pages 1567–1576, 2002.

Short Curriculum Vitae

Frank Matthias Kovatsch

Personal Data

Date of Birth	2 September 1982
Place of Birth	Erlangen, Germany
Citizenship	German

Education

2009 – 2015	**Dr. sc. ETH Zurich** Department of Computer Science, ETH Zurich, Switzerland Thesis: Scalable Web Technology for the Internet of Things
2003 – 2008	**Dipl.-Ing.** (equiv. M.Sc.) in Systems of Information and Multimedia Technology Friedrich-Alexander-University Erlangen-Nuremberg, Germany Thesis: Services for Wireless Sensor Networks – A Design for Health Monitoring and Motion Analysis
1994 – 2002	**Abitur** (general qualification for university entrance) Gymnasium Höchstadt/Aisch, Bavaria, Germany Majors: Mathematics and Physics

Professional Experience

2009 – 2015	ETH Zurich, Switzerland (research assistant)
2011	SICS Swedish ICT, Sweden (visiting researcher)
2008 – 2009	Fraunhofer IIS, Germany (graduate assistant)
2008	Fraunhofer IIS, Germany (student assistant)
2007	Siemens Pte Ltd, Singapore (intern)
2006 – 2007	Fraunhofer IIS, Germany (student assistant)
2004 – 2006	University Erlangen-Nuremberg, Germany (student assistant)
2002 – 2003	Bundeswehr, Germany (military service)
2001 – 2009	Freelancer, Germany (Web developer)