A CoAP Server with a Rack Interface for Use of Web Frameworks such as Ruby on Rails in the Internet of Things

Diploma Thesis

Henning Müller

Matriculation No. 2198830
March 10, 2015

Supervisor	Prof. Dr.-Ing. Carsten Bormann
Reviewer	Dr.-Ing. Olaf Bergmann
Adviser	Dipl.-Inf. Florian Junge

Faculty 3: Computer Science and Mathematics

Universität Bremen

2afc1e5

Henning Müller
mail@nning.io

Abstract

We present a Constrained Application Protocol (CoAP) server with a Rack interface to enable application development for the Internet of Things (or Wireless Embedded Internet) using frameworks such as Ruby on Rails. Those frameworks avoid the need for reinvention of the wheel, and simplify the use of Test-driven Development (TDD) and other agile software development methods. They are especially beneficial on less constrained devices such as infrastructure devices or application servers. Our solution supports development of applications almost without paradigm change compared to HTTP and provides performant handling of numerous concurrent clients. The server translates transparently between the protocols and also supports specifics of CoAP such as service and resource discovery, block-wise transfers and observing resources. It also offers the possibility of transparent transcoding between JSON and CBOR payloads. The Resource Directory draft was implemented by us as a Rails application running on our server software.

Wir stellen einen Constrained Application Protocol (CoAP) Server mit einem Rack Interface vor, der Anwendungsentwicklung für das Internet der Dinge (bzw. das Wireless Embedded Internet) mit Frameworks wie Ruby on Rails ermöglicht. Solche Framworks verhindern die Notwendigkeits, das Rad neu zu erfinden und vereinfachen die Anwendung testgetriebener Entwicklung (TDD) und anderer agiler Methoden der Softwareentwicklung. Sie sind vor allem auf weniger eingeschränkten Geräten wie Infrastrukturgeräten und Applikationsservern vorteilhaft. Unsere Lösung ermöglicht Applikationsentwicklung nahezu ohne Pradigmenwechsel verglichen mit HyperText Transport Protocol (HTTP) und bietet performante Handhabung von zahlreichen nebenläufigen Clients. Der Server übersetzt transparent zwischen den Protokollen und unterstützt ebenso Besonderheiten von CoAP wie Service und Resource Discovery, Block-wise

Transfers und Observing Resources. Er bietet zusätzlich die Möglichkeit, transparent zwischen JSON und CBOR Payloads zu transkodieren. Den Resource Directory Entwurf haben wir als Rails Anwendung implementiert, die auf unserer Server Software läuft.

Contents

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Basics . . . 2
  - 1.2.1. HTTP and REST . . . 2
  - 1.2.2. Ruby on Rails and Rack . . . 3
  - 1.2.3. Internet of Things and CoAP . . . 3
- 1.3. Research Questions . . . 4

2. Related Work 5
- 2.1. Frameworks . . . 5
- 2.2. Scalability . . . 6
- 2.3. CoAP . . . 6
- 2.4. DTLS . . . 7

3. Background 9
- 3.1. Concurrency and Celluloid . . . 9
- 3.2. Rack . . . 10
- 3.3. CoAP . . . 13
  - 3.3.1. Existing Implementations . . . 14
  - 3.3.2. Server-sent Updates . . . 16
  - 3.3.3. Payload Formats . . . 16

4. Objectives 19
- 4.1. CoAP Server . . . 19
  - 4.1.1. Protocol Implementation . . . 19
  - 4.1.2. Concurrency and Performance . . . 19
  - 4.1.3. Rack Interface . . . 20
  - 4.1.4. DTLS . . . 21
- 4.2. Protocol Translations . . . 21
  - 4.2.1. Headers and Payload Formats . . . 21

4.2.2. Resource Discovery and Observe . . . 21
4.3. Evaluation . . . 22

5. Design **23**
5.1. CoAP Server . . . 23
5.1.1. Rack Interface . . . 23
5.1.2. Protocol Implementation . . . 26
5.1.3. Concurrency and Performance . . . 29
5.1.4. DTLS . . . 32
5.2. Protocol Translations . . . 33
5.2.1. Block-wise Transfers . . . 35
5.2.2. Content Negotiation and Transcoding . . . 36
5.2.3. Chunked Transfer Coding . . . 38
5.2.4. Resource Discovery . . . 38
5.2.5. Observe . . . 41

6. Implementation **45**
6.1. External Libraries . . . 45
6.2. CoAP Library . . . 46
6.3. David . . . 47
6.3.1. Options . . . 48
6.3.2. Rack . . . 49
6.3.3. Protocol Implementation . . . 50
6.3.4. Concurrency and Performance . . . 52
6.3.5. Protocol Translations . . . 53
6.3.6. Architecture . . . 54
6.4. Resource Discovery . . . 54
6.5. Resource Directory (RD) . . . 57

7. Evaluation **59**
7.1. Unit Tests, Code Coverage, and Code Climate . . . 59
7.2. Benchmarking . . . 60
7.2.1. Performance . . . 60
7.2.2. Interoperability . . . 67
7.2.3. Reflection . . . 73
7.3. Developer Feedback . . . 73
7.3.1. Public Announcement . . . 74

7.3.2. Developer Observation 74

8. Conclusion **77**

8.1. Recapitulation of Results 77

8.2. Perspectives 79

A. Guides **81**

A.1. Basic Installation and Usage 81

A.2. Running the Resource Directory (RD) 82

B. Raw Evaluation Data **83**

B.1. Additional Performance Benchmark Plots 83

B.2. Performance Benchmark on Ubiquiti EdgeMAX Lite 86

B.2.1. Router Preperations 86

B.2.2. Installing Ruby and git 87

B.2.3. Using David for Performance Benchmarks 88

B.3. Developer Feedback 89

B.3.1. Public Announcement 89

B.3.2. Developer Observation 91

List of Figures **95**

List of Tables **97**

Acronyms **99**

Bibliography **111**

Acknowledgements **113**

Chapter 1

Introduction

1.1. Motivation

The Internet of Things (IoT) is an emerging technology for which a recent market study predicts a fivefold increase in connected devices in the next five years [36] and for which the Gartner Plateau of Productivity seems foreseeable [18]. With a growing maturity of this technology, methodologies and best practices will develop. This work aims to bring modern web development methods to server application software development for the IoT.

For the "classic web" using the HyperText Transport Protocol (HTTP) there are many web frameworks making the development of web applications more productive. Ruby on Rails (Rails) is one framework including features that emphasize it against other frameworks or web development approaches. It promotes the development by REST criteria, supports Test-driven Development (TDD) and implements many well-known software engineering design patterns and methods of agile development such as Active Record, Convention over Configuration (CoC), Don't Repeat Yourself (DRY) and Model-View-Controller (MVC). Additionally, the Rails community is quite vivid and there is a huge ecosystem of extensions for many recurrent problems.

Current popular IoT applications (e.g. Nest products) are often designed being unable to work without proprietary server applications provided by the vendor and jeopardize customer privacy[1]. Providing an Open Source possibility for IoT application development in a framework widely known as Rails hopefully promotes decentralization and the autonomy of users. The *Thin Server Architecture* proposed by M. Kovatsch et al. fosters application layer interoperability by shifting application logic from embedded devices to application servers [26]. Our solution aims to provide easy development of application servers.

Compared to other protocols for constrained applications, CoAP is an especially inter-

[1] https://nest.com/legal/privacy-statement

esting playground, because it addresses many shortcomings from which the Transmission Control Protocol (TCP) and HTTP suffer even in relatively unconstrained contexts. Design features such as Representational State Transfer (REST) compatibility and usage of elements like Uniform Resource Identifiers (URIs) as well as Internet Media Types make it very adaptable to HTTP.

1.2. Basics

In the next subsections we will describe the basics of the technologies that form the context of this work.

1.2.1. HTTP and REST

The HyperText Transport Protocol (see [RFC 7230] ff.) is an application protocol based on TCP. It uses URIs (`http://example.com/foo` for example) to address certain resources on different servers (see section 2.7 of [RFC 7230]). For every request of (or on) a resource, a TCP session is established with a server and a method and resource path (among other information) are sent (see section 3.1.1 of [RFC 7230]). This session can be reused for later requests. Several methods are defined in section 4 of [RFC 7231]. The most important one is probably `GET` which is used to retrieve a resource representation. The server synchronously answers with a status code that describes the result of the requested operation and the representation of the requested resource (see section 3.1.2 of [RFC 7230]). Subsequently, the TCP session is closed if there were no further requests in a timeout interval. The protocol information in the header is text based – as opposed to binary – and extended by numerous other internet standards.

Representational State Transfer (REST) is an architectural style facilitating a set of "interaction constraints" that can be applied to HTTP but also to other protocols to achieve "scalability of component interactions, generality of interfaces", to "reduce interaction latency", and "enforce security" (see section 5 of [REST]). It was defined by Roy Thomas Fielding, one of the original authors of HTTP. One of the constraints is the usage of standard HTTP methods (e.g. `GET`, `POST`, `PUT`, and `DELETE`) for interface generality. Another important constraint is statelessness which means that "each request from client to server must contain all of the information necessary to understand the request". A resource (identified by a URI) can be represented in different formats; clients are able to choose the format most suitable.

1.2.2. Ruby on Rails and Rack

Ruby on Rails "is an open-source web framework that's optimized for programmer happiness and sustainable productivity" [31]. It is designed in a way that eases web application development by REST criteria. Many well-known software engineering design patterns and methods of agile development such as Active Record, Convention over Configuration, Don't Repeat Yourself and Model-View-Controller are incorporated into the framework. Test-driven Development is well supported by Rails and widely adopted among programmers using Rails. The Rails community is quite vivid and there is a huge ecosystem of extensions for many recurrent problems.

Since version 2.3, Rails supports the Rack web server interface [Rack]. Many other Ruby web frameworks specialized on different aspects are also compatible to that interface (e.g. *Sinatra* [35], *Grape* [20]). For a more detailed introduction to Rack, refer to an article written by Christian Neukirchen (the original author) [27].

1.2.3. Internet of Things and CoAP

The Internet of Things (IoT) consists of embedded devices connected to the Internet in a wired or wireless manner. Software on embedded devices cannot rely on many resources in terms of the Central Processing Unit (CPU), Read-only Memory (ROM), and Random-access Memory (RAM). We will mostly use the term *Internet of Things*, because it is more general than other similar terms like *Web of Things* and *Wireless Embedded Internet*.

The Constrained Application Protocol (CoAP) is a protocol based on UDP specifically designed for use with constrained nodes and networks [RFC 7252] [5]. It incorporates successful design elements also used in HTTP such as URIs and Internet Media Types and aims to realize the REST architecture. In contrast to HTTP, CoAP is not necessarily synchronous. There are no long-lasting connections, and responses can be sent independently of requests. This enables features such as multicast support [RFC 7390] and server-sent notifications on resource changes (called *Observe*, see [core-observe-16]). Because of those and other features such as built-in discovery and the Resource Directory (RD) draft [core-resource-directory-02], it is particularly suited for Machine-to-Machine (M2M) communications. The *Block* extension [core-block-17] allows transfer of resource representations larger than the maximum payload of a CoAP message (about 1 KiB).

In order to make a Rack compatible application available to CoAP capable clients, it has to be translated between the HTTP centric Rack interface and CoAP. This translation has to be applied to headers, payloads, and also to more advanced protocol features.

1.3. Research Questions

The aim of this work is to answer the following question:

To which extent is it possible to use web application frameworks for HTTP such as Ruby on Rails in the Internet of Things utilizing CoAP?

More specific questions implied by the main research question can either be related to the *CoAP server* or to *Protocol translations*.

The CoAP server handles requests from the network, translates between HTTP and CoAP specifics and interfaces with the web application framework via Rack. More specific research questions are:

1. How can the Constrained Application Protocol (CoAP) be implemented?
2. How to handle a huge amount of IoT devices?
3. Does the Rack interface comply with the characteristics of CoAP?
4. How can a later integration of transport encryption with Datagram Transport Layer Security (DTLS) [RFC 6347] be supported?

Incoming requests and outgoing responses have to be translated between what the Rack interface (which is designed for HTTP) expects and the CoAP protocol specifics.

1. How can headers be translated between HTTP and CoAP?
2. How can payload formats be translated between their suitable fields of application?
3. Is the CoAP Resource Discovery (see section 7.2 of [RFC 7252]) implementable transparently for the Rack application?
4. Can a Resource Directory (RD) [core-resource-directory-02] be realized with the created software?
5. Is it possible to support Observe [core-observe-16]?

Chapter 2

Related Work

In this chapter we examine which solutions exist that partly answer our research questions and give an introduction to possible sources in that contexts.

2.1. Frameworks

Generally, CoAP server implementations provide their own Application Programming Interface (API) for applications in constrained environments to be crafted and do not support common frameworks (see a Hello World server application for *Californium*[1] and a exemplary server with several resources based on *libcoap*[2], for example). Those APIs can also be seen as the framework in which applications are developed in. Examples are *Californium* [7], *Erbium* [16] and *libcoap* [4]. For applications on constrained nodes the absence of full-featured frameworks is not remarkable, because CPU and memory resources are quite limited. For a definition of the constraints see [RFC 7228]. Applications which do not have to be especially optimized on limited resources can profit from framework utilization.

At the time of writing, we could only find one integration of a CoAP server with a common web framework. *WebIoT* [8] can be seen as a framework for constrained applications. It supports different protocols and is based on the Google Web Toolkit (GWT). However, as it is focusing on graphical applications it ships with a mandatory HTTP web interface and allows application development only as plugins. It is also not optimized for the features of a specific protocol but tries to aggregate sensor data protocol independently.

In the context of Ruby, Rack is the most widely adopted interface between web servers and applications and it is supported by many web application frameworks (see Table 3.1). The Rack specification [Rack] lists the Python Web Server Gateway Interface

[1]`https://github.com/eclipse/californium/tree/master/cf-helloworld-server`
[2]`https://sourceforge.net/p/libcoap/code/ci/master/tree/examples/server.c`

[PEP 333] as a source of inspiration. Rack is chosen for this work because of its spread and provision of a solid base.

There are many HTTP web servers written in Ruby that support the Rack interface. Although they are probably designed in a substantially different way than CoAP servers, they can be analyzed regarding to their Rack implementation and used for comparison in the performance evaluation of our work. Notable examples are *Reel*, *Thin*, and *Unicorn* (see [29], [38], and [39]).

2.2. Scalability

The inability of web servers with concurrent request handling based on processes or threads to scale up to 100.000 requests per second led to the framing of *The C10K problem* [22]. One possibility to solve this problem is event-driven architecture as it is discussed in the Reactor pattern [32] [33] designed by Douglas C. Schmidt and others. Matthias Kovatsch provides a detailed overview of web server architectures with regard to scalability in his dissertation thesis *Scalable Web Technology for the Internet of Things* [24].

For Ruby there are a number of libraries targeting the simplification of the development of concurrent programs. For this work especially *Celluloid* [9] is considered. The basics of concurrency and *Celluloid* are also highlighted in section 3.1.

Especially *Californium* is promising as a source of inspiration for performance and concurrency oriented design. According to a paper among others written by its original author Matthias Kovatsch, *Californium* handles almost 400,000 requests per second [25]. Some observations from the *Celluloid* source code are collected in section 3.3.1.

2.3. CoAP

Current and comprehensive lists of CoAP implementations are to be found in the Wikipedia [14] and on the website coap.technology [12]. Especially implementations in C/C++, Java or – of course – Ruby with a server component are qualified for a Ruby integration and therefore suitable to be based upon in the context of this work. To which extent the adaption of an already existing library is beneficial and the corresponding design questions are discussed in section 3.3 and subsection 5.1.2.

Independent of their qualification for a Ruby integration, some implementations can give valuable input regarding to the architecture and design of the basic protocol features such as deduplication (see section 4.5 of [RFC 7252]) or extensions such as blockwise transfers [core-block-17] or observe [core-observe-16]. Although this work focuses on the server side, also client implementations are interesting for their way of message parsing, usage in testing of our server implementation and their transmission layer with regards to code reuse. However, analyzing every implementation in detail would not be possible because of the time constraints of this thesis.

Both the *Cross-Protocol Proxying between CoAP and HTTP* section of [RFC 7252] (see section 10) and the *Guidelines for HTTP-CoAP Mapping Implementations* [core-http-mapping-06] can be consulted regarding to translations of CoAP headers and protocol features to HTTP and vice versa. However, simply proxying an ordinary Rails application would not suffice. An objective for our solution is that it supports a CoAP specific development and therefore should be explicit about that protocol. The mapping guidelines also make some general statements about the transparent conversion of one content format into another, called *Transcoding* (see section 6.4 of [core-http-mapping-06]).

We did not find any Open Source client or server implementations supporting the RD draft [core-resource-directory-02]. The only exception is a server implementation contained as an example in the *txThings*[3] project. However, it implements an older version of the draft and is missing functionality. The CoAP article in the Wikipedia [14] lists *CoAPSharp* as supporting the RD draft but its code contains no indications about that.

2.4. DTLS

CoAP depends on DTLS for security (see section 9 of [RFC 7252]). The DICE Working Group (WG) of the Internet Engineering Task Force (IETF) works on supporting the use of DTLS in constrained environments. The main draft that is currently worked on defines a profile of DTLS 1.2 [RFC 6347] (and Transport Layer Security (TLS) 1.2 [RFC 5246]) "that offers communication security for IoT applications" [dice-profile-09]. Another draft aims to standardize the DTLS handshake on top of CoAP to make use of block-wise transfers [schmertmann-dice-codtls-01]. There is a Ruby library[4] that implements this draft in an alpha state. An early draft [23] considers securing multicast communications in that context. The integration of DTLS into *Californium* is discussed in a master's thesis from 2012 [21].

[3]`https://github.com/siskin/txThings`
[4]`https://rubygems.org/gems/codtls`

Chapter 3

Background

For the design of different aspects of the software created during this work, some background on the server's concurrency architecture, the Rack interface, and the CoAP implementation will be given in this chapter. We will weigh the advantages and disadvantages of different concurrency models in section 3.1. The Rack interface is introduced in section 3.2. For the CoAP implementation, we analyze different existing projects to learn about implementation techniques in section 3.3. Some background will be given about the Observe extension [core-observe-16] and the Concise Binary Object Representation (CBOR) [RFC 7049].

3.1. Concurrency and Celluloid

Classicly the concurrent handling of network messages is based on processes, threads, or a combination of these. Both have some performance drawbacks through overhead regarding process state and context switching. In general, the overhead of threads is a little lower than with processes.

Using threads in Ruby (Matz's Ruby Interpreter (MRI)) even has some extra performance issues: Although Ruby threads are mapped to Operating System (OS) level threads, a Global Interpreter Lock (GIL) prevents scaling to more than one CPU per process and therefore also prohibits maximum concurrency. Threads also provide some implementation challenges such as avoiding deadlocks. Process-based concurrency tends to an even higher memory consumption, if there is a necessity to share state between processes. Kovatsch and Erb give a detailed overview of web server architectures and their advantages and disadvantages for scalability and performance in the publications *Scalable Web Technology for the Internet of Things* [24] (section 4.1.2) and *Concurrent Programming for Scalable Web Architectures* [15], respectively.

Event-driven architecture attempts to reduce the overhead caused through process or thread based concurrent handling of network messages. The Reactor pattern [32] [33] provides a pattern for implementation of event-driven architecture. Popular examples of Reactor pattern implementations in the Ruby ecosystem are *eventmachine* and *Celluloid::IO*. *Celluloid::IO* provides drop-in replacements for the socket classes from the Ruby standard library that use the reactor pattern on strictly non-blocking receive methods. It is built on top of *Celluloid* – a library that provides an "actor-based concurrent object framework" [9]. So *Celluloid* does not only provide a library for evented I/O but also a framework based on threads for object concurrency in general. *Celluloid* uses fibers for asynchronous message calls between threads. Fibers are supported since Ruby 1.9 as a manually scheduled means of concurrency that needs less memory and context changes than threads. The *Celluloid* README [10] states that "Each actor is a concurrent object running in its own thread, and every method invocation is wrapped in a fiber that can be suspended whenever it calls out to other actors, and resumed when the response is available." Celluloid is modeled after the ACTOR Formalism defined by Hewitt et al. in 1973. Actors integrate well with clean, synchronous object oriented design, because no callbacks are used for asynchronous operations. They can be supervised and automatically be restarted on crashes through unhandled exceptions for example. Linking one actor to another ensures the former also crashes or being notified when the latter fails. *eventmachine* only provides evented I/O and therefore object concurrency (application threads, messaging, etc.) would have to be approached manually when it is used. Another argument against *eventmachine* is its flawed IPv6 support [1].

The server process to be implemented can consist of several concurrent actors for different tasks such as handling new requests or managing notifications to clients observing resources. The actors handling incoming packages can utilize reactors provided by *Celluloid::IO* [11] to handle concurrent network communications.

3.2. Rack

The basic ideas of Rack are as follows. An application is a Ruby object that responds to a method named `call`. This method is invoked for example by a server when an application response is demanded. The only argument an invocation of `call` includes is a `Hash`[2] representing the environment of the request. This includes information such as the HTTP method (see section 4 of [RFC 7231]), the path, query string and HTTP headers (see section 5 of [RFC 7231]) of the request. The Rack environment was inspired by the Common Gateway Interface (CGI), an interface for web servers to retrieve HTTP

[1] https://github.com/eventmachine/eventmachine/issues/74
[2] http://ruby-doc.org/core-2.2.0/Hash.html

responses from external programs. It is also extended by Rack specific options such as the Rack interface version. An invocation of `call` has to return an `Array`[3] containing the HTTP status code (see section 6 of [RFC 7231]), a `Hash` of HTTP headers, and the response payload[4]. An example for a particularly minimal Rack application would be the Ruby code shown in Listing 3.1; normally headers such as *Content-Type* and *Content-Length* would be included with their respective values.

Listing 3.1: Minimal Rack application

```ruby
# A Proc object getting env as argument returning an Array.
->(env) { [200, {}, ['Hello World']] }
```

Rack offers the possibility to wrap the invocation of the `call` method of an application through an intermediary called *middleware*. A piece of middleware basically works like an application, also providing a `call` method with the environment as argument. The Rack application is passed upon initialization of the middleware class. On `call` invocations, the middleware is able to change the environment or the application response according to its function. As an example, an authentication mechanism would be possible to implement as a Rack middleware that returns a forbidden status code unless the request environment contains valid credentials in the query string. A popular Rack middleware example included in Rails by default and also interesting in the context of this work is *Rack::ETag* which adds entity tag headers to responses (see section 2.3 of [RFC 7232]). Rack middleware is usually independent from the framework or application, so it can be used with different frameworks supporting Rack.

Instance variable access in Rack middleware can lead to concurrency problems when conducted in a multi-threaded system, because every middleware class is only instantiated once[5]. See Listing 3.2 for a possible but rather expensive workaround.

[3] `http://ruby-doc.org/core-2.2.0/Array.html`
[4] Which has to respond to `each`, take a block and yield chunked data as `Strings`
[5] `http://stackoverflow.com/q/23028226`

Listing 3.2: Rack dup._call workaround

```ruby
class Middleware
  def call(env)
    dup._call(env)   # Duplicate instance and invoke _call.
  end

  def _call(env)       # Actual middleware logic.
    @env = env       # Safe now!
    ...
  end
end
```

There are numerous different Rack frameworks, some of them with very distinctive features. We compiled a list of frameworks and their download locations in Table 3.1 for an overview. We will refer to some of them later in other contexts.

Framework	Website
Grape	http://intridea.github.io/grape
Camping	http://camping.io
Cramp	https://cramp.in
Cuba	https://cuba.is
Brooklyn	https://github.com/luislavena/brooklyn
Hobbit	https://github.com/patriciomacadden/hobbit
Nancy	http://guilleiguaran.github.io/nancy
NYNY	http://alisnic.github.io/nyny
Ramaze	http://ramaze.net
RESTRack	http://restrack.me
Roda	http://roda.jeremyevans.net
Scorched	http://scorchedrb.com
Sinatra	http://www.sinatrarb.com
Rails::API	https://github.com/rails-api/rails-api
Ruby on Rails	http://rubyonrails.org

Table 3.1.: Rack frameworks

As the Rack interface is designed for HTTP requests and responses, some adaption has to be performed when using it with CoAP. On requests, CoAP options have to be translated to HTTP headers to be included in the Rack environment. For responses – besides this options/headers mapping in the opposite direction – also HTTP response codes have to be translated. Other functionality such as observing resources or resource discovery has to be adapted, too.

3.3. CoAP

The Constrained Application Protocol is standardized as [RFC 7252] by the IETF. It incorporates successful design elements also used in HTTP. In contrast to HTTP it is binary, based on User Datagram Protocol (UDP) by default and not bound to synchronous messaging or long-lasting connections. Its default ports are `5683/udp` for unencrypted messages and `5684/udp` when secured with DTLS. Messages can be transferred in a reliable (*Confirmable*) or unreliable way (*Unconfirmable*). Reliably transferred messages are responded to with an acknowledge message by the receiver. If a confirmable message is not responded to, it gets retransmitted in exponentially increasing intervals. A response can either occur directly (piggy-backed) or separately if generating a response would take longer than the retransmission timeout. In this case an empty message is sent first that confirms the request. When the actual response is ready, it gets sent as a new confirmable transmission. Unconfirmable messages are not retransmitted and responses are always separated.

CoAP specifies the methods `GET`, `POST`, `PUT`, and `DELETE` for retrieval, creation, update, and deletion of resources. Responses carry status codes similar to HTTP but less complex. The headers of CoAP messages can contain numerous options comparable to HTTP header fields. They are used for functionality such as Cache Validation and Conditional Requests (`ETag`, `If-Match`), or Content Negotiation (`Accept`, `Content-Format`).

As CoAP is not necessarily reliable and synchronous, it is possible to use multicast messaging to address a group of hosts with a single message[6]. This can also be used for discovery of other reachable CoAP nodes and is called *Service Discovery* (see section 7.1 of [RFC 7252]). A special resource with the path `.well-known/core` is specified to return information about the available resources of a node. This procedure is referred to as *Resource Discovery* (see section 7.2). As the maximum payload size of a CoAP message is around 1 KiB, larger resource representations have to be transmitted in several messages as it is defined in an extension for *Block-wise Transfers* [core-block-17]. Another

[6]Although messages effectively are broadcasted on some physical layers such as [IEEE 802.15.4]

extension named *Observing Resources* specifies a way to register for and subsequently receive updates about the state of a resource [core-observe-16].

3.3.1. Existing Implementations

In the context of the research questions, it is interesting whether an implementation supports block-wise transfers [core-block-17], observe [core-observe-16] and (multicast) group communications [RFC 7390]. Table 3.2 lists Open Source implementations obtained from the CoAP article in the Wikipedia [14] and the coap.technology website [12] that support server mode, block-wise transfers and observe. The coap.technology website lists a pure Ruby implementation developed in the student project *GOBI* of the communication networks research group (AG Rechnernetze[7]) at the Universität Bremen based on a CoAP message parser by Carsten Bormann. Information about compatibility with Ruby and group communications support is added to the compiled list. A library is seen as compatible if it is written in Ruby itself, C/C++ or Java. Whether multicast communications are supported has been determined by analyzing the respective source code and searching for group addresses specified by section 12.8 of [RFC 7390]. There are libraries like *libcoap* that are multicast compatible but do not listen on CoAP group addresses by default or leave it to the programmer to do so. *CoAPthon* and *SMCP* are the only implementations listening on the CoAP group addresses by default[8].

Library	Compatibility	Multicast ([RFC 7390])
Californium	✓	×
CoAPthon	×	✓(IPv4)
COAP.NET	×	×
Erbium for Contiki	×	×
jcoap	✓	×
libcoap	✓	×
node-coap	×	×
Ruby coap (GOBI)	✓	×
SMCP	✓	✓(IPv6)
txThings	×	×

Table 3.2.: CoAP Libraries – Possible compatiblity with Ruby and multicast support

[7] `https://ag-rn.tzi.de`

[8] We also tested incompatible implementations.

In the following paragraphs, we briefly examine the source code of *Californium* and *libcoap* with regards to their architecture, implementations of the CoAP Message Deduplication (see section 4.5 of [RFC 7252]) and concurrency approaches. Table 3.3 lists the examined versions.

Library	Version
Californium	1.0.0-M3 (Core)
	1.0.0-M3 (Connector)
libcoap	`d48ab44`

Table 3.3.: CoAP Libraries – Examined versions

Architecture

Some architectural inspirations will be mentioned but we will not analyze the overall architecture of every library.

Californium uses an approach similar to the design of Rack to implement different functional protocol "layers" (see line 83 of `CoAPEndpoint` class (Core)). Messages are passed through a stack of classes each handling a certain transmission layer task. A `ReliabilityLayer`, for example, retransmits messages if there was no response; a `BlockwiseLayer` handles block-wise transfers.

Message Deduplication

Californium makes use of the Java class `ConcurrentHashMap` as a hash table for managing data about ongoing transmissions (see line 58 to 61 of `Matcher` class (Core)). Three hash tables are used with different keys (message ID, token, and URI). The Java documentation describes `ConcurrentHashMap` as a "hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates" [13].

Concurrency

Californium maintains a pool of threads which send or receive messages (see `UDPConnector` class (Connector)). Other parts of the code asynchronously schedule messages to be sent (see for example line 178 of `ReliabilityLayer` class (Core)). *Californiums* use of `ConcurrentHashMap` ensures safe and performant concurrent access to data shared by several threads.

libcoap itself does not implement a server ready to use but provides the API for programmers to base their own server implementations on. The source code contains an example server that is realized as a single threaded event loop (see line 414 of `examples/server.c`).

3.3.2. Server-sent Updates

In the HTTP world, some protocols are designed to overcome the strictly synchronous request model of HTTP/1.1 (see [RFC 7230] ff.) and are therefore in some extent comparable to the CoAP Observe extension [core-observe-16]. Their properties differ in distinct ways from Observe, which is not keeping connections open, is unidirectional and does not strictly require the server pushing updates only directly after requests of the client. *Long Polling* and *Streaming* are two mechanisms common to achieve server-initiated communication with HTTP. [RFC 6202] summarizes known issues and lists best practices regarding to the use of these mechanisms. Popular choices for this purpose in the traditional web are Comet[9] and WebSockets [RFC 6455]. Both use long opened connections, which puts extra load on the servers. Comet just keeps a HTTP GET request opened, which is used by the server to push updates gradually. WebSocket specifies bi-directional out-of-band (non-HTTP) communications. Solutions more similar to Observe are HTTP/2 Server Push (see section 8.2 of [httpbis-http2-17]) and HTML5 Server-Sent Events [W3C REC eventsource], which both are unidirectional but also employ long-running connections. HTTP/2 Server Push is not truly asynchronous, because the push mechanism is only used to multiplex documents anticipating a direct request. HTML5 Server-Sent Events are most similar to Observe from an architectural perspective. The implementation of the mentioned protocols in connection with Rack and Rails can give valuable input for the design of the Observe integration in the context of this work. Therefore some existing Ruby gems that implement the mentioned techniques can be a source of inspiration. We further analyze that topic in subsection 5.2.5.

3.3.3. Payload Formats

There are a number of different formats for resource representation (serialization) that are more or less optimized on low resource usage (when transmitted or parsed). In the world of less constrained web applications especially JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) are common as resource representations for APIs. Both are convertible into less resource demanding binary serialization formats.

[9] `https://en.wikipedia.org/wiki/Comet_(programming)`

For JSON there are for example Binary JSON (BSON), Universal Binary JSON (UBJSON), MessagePack and CBOR. XML can be converted into Efficient XML Interchange (EXI) to save resources. We will concentrate on JSON, because it currently is more widespread than XML in the context of modern web application development. Especially CBOR is considered as a binary serialization format due to its closeness to JSON, the compactness (of code and data), and the possibility to be used without a schema description (see section 1.1 of [RFC 7049]).

Chapter 4

Objectives

Corresponding to the primary research question stated in section 1.3, the higher objective of our work is to determine the possibility of using Rack based web frameworks originally designed for HTTP in the IoT utilizing CoAP. We utilize Test-driven Development (TDD) for the software components. There are also some general design goals to the code such as code quality, modularity, and sustainability through readability and a high test coverage for example.

4.1. CoAP Server

4.1.1. Protocol Implementation

As already shown in subsection 3.3.1, there are numerous different implementations of the Constrained Application Protocol (CoAP) in many programming languages and with differing design goals. We will examine some of them to determine the implementation to base the server developed during this work on. The support for a recent standard version [RFC 7252] and extensions for observing resources [core-observe-16] and block-wise transfers [core-block-17] is particularly important. We will analyze and design different aspects of the protocol such as Message Correlation and Deduplication as well as different response types from a server's perspective and implement them accordingly. General design goals are the robustness and interoperability of the CoAP implementation.

4.1.2. Concurrency and Performance

For the handling of frequent sensor data updates from a huge number of IoT devices, the concurrency model has to be chosen fittingly. Concurrent handling of requests is therefore one design goal of the server implementation. We will analyze which models are viable for applications in the IoT and how possible models integrate with Rails. Neither the server itself, nor the CoAP implementation or the application framework may block the concurrent handling of further requests. For a realistic objective for the concurrency level and the maximum number of requests per second of the implementation, benchmarks of other implementations should provide for an overview. It is important to consider the comparability of different benchmarks, as they differ at least on two levels: First, the hardware configurations range from ultra-portable, energy-saving developer notebooks, to several full-fledged, high-end servers. Secondly, the choice of frameworks for testing is important. A benchmark testing a plain Hello World Rack application is not comparable with its Rails or *Sinatra* counterpart, because of the framework overhead. For this work, only the server performance has to be measured, so benchmarking different web servers on several Ruby VMs serving a plain Hello World Rack application would be of most value. The server implemented during this work does not have to scale as well as *Californium* for example, which according to a paper among others written by *Californiums* original author Matthias Kovatsch handles almost 400,000 requests per second [25] via CoAP on server hardware. For Ruby applications, those values are reachable over HTTP with server stacks not written in Ruby such as TorqBox on JRuby or nginx proxying Unicorn on MRI for example[1]. With HTTP web servers written in Ruby running plain Rack applications (no further frameworks like Rails) scale up until approximately 7.600 requests per second[2], so it is expected that with CoAP this number could be improved in comparison to HTTP. For a server scaling this well still much optimization work has to be done, which is not the focus of this work. The more conservative number of 5.000 requests per second shall be aimed at, for now. Implicated objectives by these performance goals are the portability of the server and the CoAP implementation to other Ruby Virtual Machines (VMs) than MRI, such as JRuby for example, and the choice of a server concurrency model which scales well with the provided hardware.

4.1.3. Rack Interface

As many other Ruby web frameworks, Rails interacts via the Rack interface with the web server. The main targeted web application framework will be Rails but the server implementation shall also be tested against different other Rack middlewares and application frameworks. It shall be clarified in which extent this interface suffices for the characteristics of the Constrained Application Protocol (CoAP). The translation between the

[1] `http://www.techempower.com/benchmarks/#section=data-r9&hw=peak&test=json&l=35s`

[2] `http://www.madebymarket.com/blog/dev/ruby-web-benchmark-report.html`

Rack interface based on HTTP and CoAP has to function seamlessly and be as complete as possible.

4.1.4. DTLS

The server has to be designed to allow the integration of DTLS [RFC 6347] as a means of transport encryption. We do not pursue an actual implementation. The implications of DTLS in the context of this work shall be examined. Functionality such as observe or certain Rack middleware might possibly be derogated.

4.2. Protocol Translations

4.2.1. Headers and Payload Formats

The software implemented during this work shall enable an integrated development of applications for the traditional, human-readable web and machine-to-machine communication in the IoT. It will be clarified how the HTTP and CoAP Content Negotiation protocols (see section 5.3 of [RFC 7231] and 5.5.4 of [RFC 7252]) can be utilized for the purpose of separating those client groups from a developers perspective in Rails. Other functionality of HTTP based upon headers applicable to CoAP shall also be preserved using an appropriate translation of the headers (for example the `ETag` header which is defined in section 2.3 of [RFC 7232]). Also the implications of a transparent content type mapping for responses by the web framework between a data serialization format of the traditional web and more resource-saving ones for the IoT will be emphasized. A transcoding of resource representation formats between JSON [RFC 7159] and CBOR [RFC 7049] will be implemented.

4.2.2. Resource Discovery and Observe

Furthermore, special properties of CoAP such as the Resource Discovery (see section 7.2 of [RFC 7252]) and Observe [core-observe-16] will be considered and integrated appropriately. The Resource Discovery shall be integrated into the web framework (for example Rails) without any developer interaction for resources for which a interaction with CoAP is reasonable. The functionality of an RD [core-resource-directory-02] shall be facilitated by pre-implemented, configurable, and well-tested modules. It will also serve as application to demonstrate the implementation properties. In order to support Resource Discovery and the implementation of an RD, Group Communication for the CoAP [RFC 7390] will be integrated. Additionally, a way to integrate the asynchronous nature of Observe and the rather synchronous ones of Rack and Rails will be designed and implemented.

4.3. Evaluation

For the evaluation, it shall be analyzed to which extent the specified goals of functionality are met, how the performance of the implementation compares to related work and how Rails developers respond to given tasks in connection with the developed software modules.

Unit tests emerge during the Test-driven Development (TDD) that cover most parts of the implementation source code and that specify and inspect the functionality of individual modules, classes and methods of the different components. The coverage of source code by tests (*Code Coverage*) will be measured. For the demonstration application, unit, functional and integration tests shall be created that besides testing the application itself also demonstrate TDD on the covered software stack with methods common in Rails. The CoAP library will be tested for its interoperability with other implementations in an automated way.

The CoAP server will be compared to other implementations by performance tests. In doing so we will also measure the consumption of resources. We will also compare the performance of different Rack based frameworks in connection with the server. For the applicability of the complete stack for the usage of control nodes in home automation networks, a test installation will be made on home router hardware. The data gained by the performance tests will be used to reflect on design decisions as the kind of solution for the realization of the CoAP server.

To evaluate the developer friendlyness and the seamlessness of the integration in Rails, experienced Rails developers are going to be interviewed after they solved a given task. This task shall contain CoAP specifics that the developers do not necessarily know of.

Chapter 5

Design

The main component being developed through this work is a CoAP server able to serve responses from frameworks compatible to the Rack interface. Section 5.1 discusses interfaces, libraries and the concurrency model, and section 5.2 analyzes translations between the HTTP and CoAP protocols.

5.1. CoAP Server

The code will be modularized with mixins (Ruby modules included in classes), because these need less memory compared to the instantiation of classes. Where tuples of data are needed (as keys for the message deduplication cache, for example), structs were considered instead of dynamically growing types like arrays. However, a performance advantage of structs could not be measured (see `experiments/structs.rb` of the david repository). On MRI, both Arrays and Structs are implemented in C.

5.1.1. Rack Interface

The stable Rack version used in development is 1.6.0[1]. As a source of the Rack interface specification the version from the current master branch [Rack] is referenced here, because it was not possible to find a readably formated version of the specification for Rack 1.6.0.

From the servers perspective, on the Rack side only the environment has to be provided correctly and completely and the application's `call` method has to be invoked with that environment on incoming requests. The `call` invocation returns the described `Array`, which contains values specific to HTTP and therefore has to be translated to CoAP. For details on the translation of incoming CoAP messages and outgoing Rack responses see "Protocol Translations" (section 5.2) on page 33. There are Rack

[1] `https://rubygems.org/gems/rack/versions/1.6.0`

Listing 5.1: Exemplary Rack environment

```
{
  'REMOTE_ADDR'        => '::1',
  'REMOTE_PORT'        => '41414',
  'REQUEST_METHOD'     => 'GET',
  'SCRIPT_NAME'        => '',
  'PATH_INFO'          => '/.well-known/core',
  'QUERY_STRING'       => 'href=test',
  'SERVER_NAME'        => '::',
  'SERVER_PORT'        => '5683',
  'HTTP_ACCEPT'        => 'application/json',
  'rack.version'       => [1, 2],
  'rack.url_scheme'    => 'http',
  'rack.input'         => #<StringIO:0x007f3d48021570>,
  'rack.errors'        => #<IO:<STDERR>>,
  'rack.multithread'   => true,
  'rack.multiprocess'  => true,
  'rack.run_once'      => false,
  'rack.logger'        => #<Logger:0x007f3d644e52e8>
}
```

specific environment options which are used to pass in objects for input, errors, or for *hijacking* the socket (see "Observe" (subsection 5.2.5) on page 41) and options for concurrency for example. An exemplary Rack environment is given in Listing 5.1 as a Ruby Hash. A more detailed description of the requirements of the Rack environment is given in its specification [Rack].

Choosing the server when running from Rack configuration files (`config.ru`) or when starting the server in a Rails project for example is possible by providing a Rack handler for the CoAP server and registering it. After registration, the desired handler can be chosen with a parameter to `rackup` or `rails server`. A Ruby code example of a basic Rack handler prototype is shown in Listing 5.2. The handler from the example would be started by `rackup -s example` or `rails s example`.

The `.well-known/core` interface (see [RFC 6690] and section 7.2 of [RFC 7252]) could be implemented as a Rack middleware. A default Rails application contains already numerous Rack middlewares for many possible purposes, some of them not applicable to CoAP based communications. `ActionDispatch::Cookies` for example manages HTTP Cookie handling. For now, we do not provide any CoAP features that

Listing 5.2: Rack handler prototype

```ruby
# The Rack library is required.
require 'rack'

# Namespace for register method.
module Rack::Handler
  # The custom handler class.
  class ExampleHandler
    def self.run(app, options = {})
      # Blockingly start actual server..
    end
  end

  # Register handler class as 'example'.
  register(:example, ExampleHandler)
end
```

are translated to HTTP using Cookies. It has to be evaluated if the removal of this middleware actually breaks code which might be useful in the context of constrained application development. `ActionDispatch::DebugExceptions` renders exceptions as HyperText Markup Language (HTML), which is way to verbose for development with a CoAP client. Both these examples and other middleware are not needed. Therefore a way has to be found to deactivate unessential middleware in a Rails CoAP application. With a component called Railtie [28] it is possible to run code from a gem upon initialization of a Rails application and to provide configuration options which can be used from inside the applications configuration files. The middleware cleanup can be configurable through a Rails option. A middleware catching exceptions thrown in a Rack application and converting them to a description for the programmer (JSON for example) would support the development process. For Rails, such a middleware exists with `ActionDispatch::ShowExceptions` but it does not support including exception details in the JSON body.

A `Rack::Lint` class instance can be inserted into the middleware stack to automatically test the Rack environment for conformity to the Rack specification [Rack].

5.1.2. Protocol Implementation

The Rack interface has to be pure Ruby code in any case, so it is suggested to also use a protocol implementation integratable into Ruby. Compared to other languages, it is possible to include C/C++ and Java code easily and without indirections like socket communications and serialization to connect a non-Ruby component with a Ruby one. Table 3.2 gives an overview of compatible CoAP libraries in different languages. So even though the server implementation in another language could promise performance benefits (see subsection 5.1.3), a Ruby solution is preferred here. Both C/C++ and Java tie the Ruby VM down to a solution written in the respective language. Using a CoAP implementation in C/C++ would require to write a wrapper C extension which uses the Ruby API to expose the functions to Ruby code. This makes the usage in JRuby impossible, since as of JRuby 1.7 the C extension support is deprecated[2]. On the other hand, a CoAP implementation in Java will not run in any Ruby VM outside of a Java VM.

A pure Ruby implementation of CoAP can be based upon an existing message parser written by Carsten Bormann, as it was done in the *SAHARA*[3,4] and *GOBI*[5] projects of the communication networks research group (AG Rechnernetze[6]) at the Universität Bremen. The *GOBI* implementation has been published on github.com[7] and features client functionality, basic block-wise transfer and observe support. But it requires some refactoring work to meet the objectives of this work such as code quality and non-existence of side effects for concurrency support. The Code Climate report[8] on the library lists highly complex classes and much code duplication (both indicating a bad structure), and many "code smells" (indicating non-conformance to the basic commonalities of Ruby style guides [3] for example). As of the current commit (1cd1244) in its git repository, the code climate scores 0.7 (with 4.0 being the best possible result) and the test coverage as measured by coveralls.io amounts to 88.19%. Arguments for forking this library and developing it further under this name are that it is already present on rubygems.org as *coap* and that it enables quick prototyping. Although we can not be sure, the code developed during this thesis can be merged back again into the original project.

[2] https://twitter.com/headius/statuses/281091403919003649
[3] https://sahara.tzi.org
[4] http://www.informatik.uni-bremen.de/cms/detail.php?id=75116
[5] http://gobi.tzi.de
[6] https://ag-rn.tzi.de
[7] https://github.com/SmallLars/coap
[8] https://codeclimate.com/github/SmallLars/coap/code?sort=remediation_cost&sort_direction=desc

As a CoAP **server** implementation is needed in the context of this work, one of the implementation goals regarding the fork of this library is making it usable in the context of a server. Abstraction and modularization to enable code reuse and testability are methods to accomplish that. The current client behavior is contained in one huge method that is called recursively in some cases. One case, for example, is the assembly of single block-wise CoAP messages. There are recursion limits to hinder huge resource consumption that are also preventing more than 10 outgoing and more than 50 incoming blocks. Sending, receiving, retransmissions, parsing, reassembly of block-wise transfers, and other protocol states or operations can be organized in a transmission layer (documented for example as finite state machines in the CoAP Implementation Guidance draft [lwig-coap-01]) that is reusable by the server, thus avoiding code duplication. Same goes for other common functionality such as utilities for handling block-wise transfer [core-block-17]. A more close implementation of the CoAP protocol is a further implementation goal that has several advantages. By improving matching and generation of message IDs and tokens, security and reliability can be enhanced. With avoidance of computation on bogus messages, application state both on the server and client side can be minimized. The API of the client is planned to be cleaned up and simplified. For example, the port argument should be optional for requests; obviously, the CoAP default port can be used, if the argument was omitted. As the server aims for portability between Ruby VMs, also the common library and client component have to support them. *Celluloid::IO* [11], which is described more detailed in the following section, can not only be used for the server, but also for the client to realize non-blocking socket operations. To further support concurrency, the client and common library code will be revised regarding to globally shared state and side effects. Tests for the server component are intended to be implemented utilizing RSpec [30]. New tests for the client component will also be written in RSpec and some of the old tests will be ported during the development. For the client, test goals consist of increasing the test coverage (for example through modularization), decoupling of existing tests from remote hosts, and increasing testing speed. Only optional integration or protocol implementation compatibility tests should remain dependent on remote hosts. For the Resource Discovery (see section 7 of [RFC 7252]) both on server and client side, the Constrained RESTful Environments (CoRE) Link Format [RFC 6690] which describes (available) CoAP resources is necessary to be implemented.

Duplicate Detection and Message Correlation

The server will send cached response messages on requests identified as duplicates (see section 4.5 of [RFC 7252]). Old entries in the response cache have to be "garbage collected". The cache used can also be utilized for the correlation of answers to sent messages. As the message receiving, parsing and dispatching happens in a single threaded event loop, the cache can be accessed without locking. However, other threads such as the garbage collector for example need to access the cache concurrently. This can be accomplished in a thread safe manner by using messaging among actors. A simple implementation of this cache could utilize a Ruby Hash with endpoint address and message ID as keys and the message and a timestamp for retransmissions and garbage collection as values. Optimizations are discussed in subsection 5.1.3. Retransmission of `CON` messages that received no `ACK` or `RST` answer in a certain time can be handled by another thread operating on the mentioned cache. Additionally, the cache has to include a timeout and a retransmission count per entry to keep track of retransmissions (see section 4.2 of [RFC 7252]). However, if the server does not send any confirmable messages, it does not have to perform retransmissions.

Separate Responses

If the response to a confirmable request takes longer than the retransmission timeout of the client, separate responses can be used to prevent a retransmission (see section 5.2.2 of [RFC 7252]). Support for separate responses requires a possibility to run application code in the background and to later send the resulting response message. When using Rack with HTTP, the server has to keep a connection alive and later send an asynchronous response. Therefore this response has to be passed to the server. Some methods to pass an asynchronous response to the server have been developed in the Rack community. *Rack hijack* (see subsection 5.2.5) is the only one being part of the Rack specification [Rack]. Providing a Proc-like as value to the Rack environment key `async.callback` is another method that was integrated into *Thin* [38]. Through `throw :async` or a special Rack response (`[-1, {}, []]`) it is signalled that the response is obtained via the `async.callback`. Another method implemented in *Thin* is a "deferrable body" object[9]. In each of these cases, the middleware used must be aware or is otherwise circumvented[10]. *Rack hijack* suffers from the same problem. *Reel* – interesting because also based on *Celluloid* – does not have a solution to this problem that works with the Rack interface.

[9] https://github.com/macournoyer/thin/blob/v1.6.3/example/async_app.ru#L65
[10] https://github.com/rkh/async-rack/blob/v0.5.1/README.md

Listing 5.3: Seperate Response

```
class ThingsController < ApplicationController
  def show
    separate do
      render json: Thing.find(params[:id])
    end
  end
end
```

As CoAP is not constrained to keep a connection alive, there is another way of implementing separate responses that is also mentioned in section 5.2.2 of [RFC 7252]. Before initiating the generation of a response by the application, the server can start a timeout. When the timeout finished before a response is available, the server sends an empty `ACK` message indicating a separate response. The actual response generated by the application then has to be sent as a confirmable or non-confirmable message. A method can be provided with which an application programmer can immediately signal a separate response if a long response time is foreseeable.

Listing 5.3 shows an exemplary source code from the application programmer's side. The call of the `separate` method would send an immediate response to the client by messaging the server actor. To actually asynchronously run the code, the block and the controller instance is passed to a separate *Celluloid* actor. This single actor or actor from a thread pool would then `call` the block in the context of the controller instance. The instance would have to persist until the asynchronous code finished running. The controller response would have to be translated like it is in a piggybacked case. The timeout method mentioned in the previous paragraph enables also the separate answer passing the Rack middleware stack but the generation of the response is happening synchronously. For the `seperate` method extension it has to also be ensured, the response is passing the Rack middleware stack.

5.1.3. Concurrency and Performance

To meet the performance objectives of this work (as stated in subsection 4.1.2), some consideration is necessary. As layed down in section 3.1, we will use *Celluloid* [9] as concurrent object "framework" and the *Celluloid::IO* [11] reactors for receiving from sockets.

There are different web server architectures. We choose a Singe-Process Event-Driven (SPED) architecture [6] for its reasonable performance and scalability as well as its ease

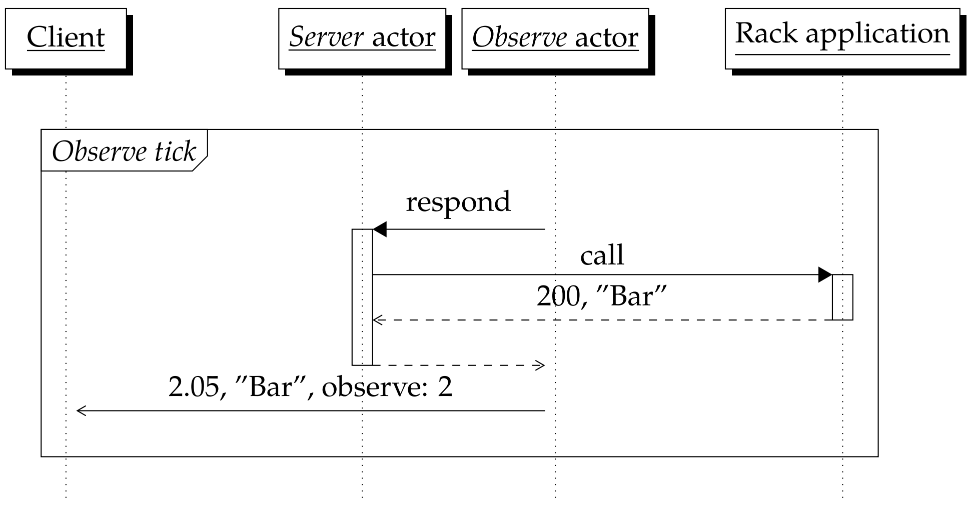

Figure 5.1.: Exemplary actor interaction on *Observe tick*. The *Observe* actor receives the Rack application response from the *Server* actor and subsequently notifies the client about a change of the resource representation.

of implementation. Some experiments were conducted with different message handling models that are documented in `experiments/concurrency/stub.rb` inside the David source code repository. Message handling, parsing, and dispatching happens inside an asynchronous event loop run from a *Server* actor of which a single instance (thread) is existent. *Celluloid::IO* abstracts the I/O selection independent from the OS using *nio4r*[11]. We avoided the frequent use of asynchronous method calls with *Celluloid's* `async` through Fibers (for example for the dispatch method) as it is creating an overhead with every request. The Rack framework call to get a response to a request also happens inside the main event loop. With this way of message handling it is important that the framework or the application programmer does not do blocking I/O operations, because they stop other requests being processed. In the Rack application or framework, non-blocking database adapters have to be used. Rails for example ships *ActiveJob* as a component to easily integrate asynchronous operations since version 4.2.

Another use case for a *Celluloid* actor is the management of observe [core-observe-16] register, deregister and notification operations. In Figure 5.1, an exemplary communication flow between the *Server* and *Observe* actors is shown. The graph describes actions performed to handle observe notifications in a certain interval. For a more detailed view on the design regarding observe, refer to "Observe" (subsection 5.2.5) on page 41.

[11] `https://github.com/celluloid/nio4r`

The *Observe* actor calls the Rack application to determine if the resource representation changed. This happens from the *Observe* actor's thread concurrently to the *Server* thread. The framework and the application programmer have to ensure that every data access is thread safe. Rails itself is capable of multiple threads concurrently calling for a framework response. However, version 4.2 uses a Rack middleware called `Rack::Lock` that locks out concurrent framework response calls. This can be deactivated with the Rack environment option `rack.multithread` set to `true`. We will deactivate `Rack::Lock` and allow concurrent framework response calls by default in the server software.

A *GarbageCollector* actor removes obsolete entries from the message deduplication cache and the list of observers (see subsection 5.2.5). The cache is placed as a Hash instance variable inside the *Server* actor and can be accessed from other actors through messages. This ensures thread safe retrieval and update of cache entries. Performance tests during the development of the concurrency architecture of the server software showed that accessing the cache through *Celluloid* actor messaging has to be used thrifty. Placing the cache in a separate actor and performing the main lookups and cache inserts via messaging from the *Server* actor had a negative performance impact of over 500 requests per second (presumably through the messaging overhead). In JRuby 1.7 this overhead is even bigger since it implements fibers through OS threads[12]. In MRI (at least until 2.3) the standard library Hash class neither is thread safe nor does it support concurrent retrieval or update [37]. With a GIL assuring only one thread running at a time at least thread safety is not a problem [2]. JRuby supports several threads running concurrently and implements updating instance variables in a thread safe manner[13]. So in both MRI and JRuby, the standard library Hash class should suffice when it comes to conflict free concurrent access. As described in section 3.3.1, *Californium* uses a Hash class supporting concurrent retrieval and updates for performance reasons. Adopting a concurrent Hash class for Ruby may also optimize performance. The *thread_safe* gem[14] provides different thread safe data structures that are Ruby VM independent. It includes a hash-like object called `ThreadSafe::Cache` which according to the README file has "much better performance characteristics esp. under high concurrency". However, we could not measure a performance gain compared to the Ruby standard library Hash (see `experiments/thread_safe.rb` in the server's respository[15]). As structures, both a Hash with two element Arrays (endpoint and message ID) as keys, and a nested Hash (`{endpoint => {mid => value}}`) were considered. Although the lat-

[12] https://github.com/celluloid/celluloid/wiki/Gotchas#tasks--jruby
[13] https://github.com/jruby/jruby/wiki/Concurrency-in-jruby#thread-safety
[14] https://github.com/ruby-concurrency/thread_safe
[15] https://github.com/nning/david

ter performs better in a detached experiment (see `experiments/hash_key.rb`), we could not measure performance benefits in the overall benchmarking of the server (see the `mid_cache_nested_hash` branch). According to the analysis of MRI 2.2 in *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals* the performance of arbitrary objects as Hash keys depends on the quality of their `#hash` instance method (see Experiment 7-3 of [34]). Like other objects, Arrays inherit a well-working `#hash` method from the *Object* class.

To keep the overall object count and therefore also the Ruby garbage collection overhead low, some objects such as `Strings` containing HTTP header names for example are instantiated as frozen `String` constants at load time.

If the Rack environment key `rack.multithread` is set to true – as we intend to do by default – the middleware `Rack::Lock` no longer ensures only one thread invoking `call` on the middleware stack. Rack middleware classes writing to instance variables are not necessarily thread-safe, because they are only instanciated once per process. This is easily fixed by using the *dup._call* workaround. The original `call` method is renamed to `_call` and a new `call` method invokes `_call` only on previously duplicated instances of the middleware class. We will use this method on any implemented Rack middleware.

5.1.4. DTLS

The component abstracting transmissions outlined in "Protocol Implementation" (subsection 5.1.2) on page 26 has to be designed in such a way that the socket class (`UDPSocket` or with *Celluloid::IO* `Celluloid::IO::UDPSocket` for unencrypted communication) instantiated for communications can be configured. A DTLS implementation would have to provide an API like the Ruby `UDPSocket` class. By default, `Celluloid::IO::UDPSocket` is used, because it is event-driven. To preserve this functionality, a drop-in socket class replacement with DTLS support would have to be modeled after `Celluloid::IO::UDPSocket`. This way, a server with a socket abstracting DTLS encrypted packets can be started instead of or concurrent to the server used for unencrypted messages.

Unlike with HTTP, with CoAP it is not unlikely an identity derived from a device's public key or a certificate is used to authenticate a peer (see section 9 of [RFC 7252]). If the Rack application is suitably implemented, it can adopt this identity assuming that it is passed in through the Rack environment. A Rack environment key such as `coap.dtls.id` could contain the identity in *RawPublicKey* mode or the certificate in *Certificate* mode. Also the Rack environment can provide a key (for example `coap.dtls`) which can be used for the app to determine the DTLS encryption mode.

Transparent authentication based on DTLS would require the authentication relationship to be terminated at the server and translated into a Rails authentication framework specific (e.g. warden[16]) `Rack::Session` component. It would also require a mapping between DTLS identities or certificates and application specific subjects which has to be provided by the application itself.

5.2. Protocol Translations

To translate between CoAP and the HTTP centric Rack interface, the specifications for *Cross-Protocol Proxying between CoAP and HTTP* (see section 10 of [RFC 7252]) and the *Guidelines for HTTP-CoAP Mapping Implementations* [core-http-mapping-06] can be used. However, a CoAP-HTTP proxy in front of an ordinary HTTP Rails application would not suffice the objectives of this work as the framework shall support a CoAP specific development and therefore should be explicit about CoAP. The Cross-Protocol Proxying section states that "since HTTP and CoAP share the basic set of request methods, performing a CoAP request on an HTTP resource is not so different from performing it on a CoAP resource" (section 10.1 of [RFC 7252]) and defines a few departures of behavior of the methods GET, POST, PUT and DELETE that have to be implemented accordingly. Section 7 of [core-http-mapping-06] defines a mapping of CoAP to HTTP response codes (also see section 6 of [RFC 7231]). That, however, can not be applied in this case, because the framework's HTTP response code has to be mapped to a CoAP response code. Tables 5.1 and 5.2 define mappings in the opposite direction (adopting entries from the CoAP to HTTP mapping table where possible). As a fallback, HTTP return codes not present in the mapping table are translated via division by 100 (e.g. HTTP *404 Not Found* maps to CoAP *4.04*). To enable explicit returns of CoAP response codes from Rack applications, `Float` values can be used. In this case, the mapping has to be skipped.

Informational HTTP response codes (as defined in section 6.2 of [RFC 7231]) do not result in an immediate answer by the server and are therefore not translated. As CoAP supports no *Redirect* HTTP response codes, another way has to be found to pass redirection information to the client. It would be intransparent for the server to follow the redirects and return the redirection target resource representation to the client as if there was no redirection. The client would have no way to determine if a resource is actually

[16]https://github.com/hassox/warden

HTTP	CoAP
200 OK	2.05 Content
202 Accepted	2.01 Created
203 Non-Authoritative Information	2.05 Content
206 Partial Content	Not applicable
207 Multi-Status	Not applicable
208 Already Reported	Not applicable
226 IM Used	Not applicable

Table 5.1.: Mapping of HTTP to CoAP return codes (*2xx Success*)

existent. To pass the redirection information to the client, the server can translate the Rack redirect answer into a JSON document. An Example is given in Listing 5.4. If conditional requests (see section 5.10.8 of [RFC 7252] and 3 of [RFC 7232]) are implemented, HTTP *304 Not Modified* would be mapped to CoAP *2.03 Valid*. All *Server Error* response codes that are not translated directly via division (505, 506, and 511) map to the generic CoAP *5.00 Internal Server Error*.

Listing 5.4: Exemplary JSON redirect description

```
{
  "code": 302,
  "location": "/example"
}
```

To return CoAP codes as accurate as possible, sometimes not only the HTTP return code but also the request method has to be considered. An example for this case is the answer to a CoAP DELETE request. Section 4.3.5 of [RFC 7231] states "the origin server SHOULD send a 202 (Accepted) status code if the action will likely succeed but has not yet been enacted, a 204 (No Content) status code if the action has been enacted and no further information is to be supplied, or a 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status". The CoAP specification (see 5.8.4 of [RFC 7252]) only demands a CoAP *2.02 Deleted* status code independent of the success cases. So for a more correct implementation, response codes have to be mapped from a tuple of HTTP method and response code to a CoAP response code.

Where a direct translation is adequate, the basic mapping of headers will be imple-

HTTP	CoAP
402 Payment Required	Not applicable
407 Proxy Authentication Required	4.01 Unauthorized (Not really applicable)
409 Conflict	4.12 Precondition Failed
410 Gone	4.04 Not Found
411 Length Required	4.02 Bad Option
414 URI Too Long	4.02 Bad Option
416 Range Not Satisfiable	4.02 Bad Option (Not really applicable)
417 Expectation Failed	Not applicable
426 Upgrade Required	Not applicable

Table 5.2.: Mapping of HTTP to CoAP return codes (*4xx Client Error*)

mented as shown in Table 5.3. (This table does not consider all existing HTTP headers, because of their huge number.)

Some translations could be solved through Rack middleware (see subsection 5.1.1). Except for the Resource Discovery, translation through methods present directly in the server code is favored over middleware usage. Using middleware for translating HTTP to CoAP headers would change the Rack environment to contain CoAP headers. This would break transparency of that exemplary middleware and it will not be possible to prepend it by another middleware expecting HTTP headers.

The following subsections describe the translation of more complex protocol features.

5.2.1. Block-wise Transfers

The most naive implementation of handling requests for certain blocks of a resource representation through block-wise transfers [core-block-17] would be requesting a full framework answer on each request and returning only the requested chunk. However, if the resource representation changes in between different requests with Block2 Option in control usage (see section 2.3 of [core-block-17]), the payload possibly can not be reassembled on the client side. According to section 2.4 of [core-block-17] a server should include an ETag option with its responses. This way a client can try to obtain a fully consistent resource representation again if the ETag value suddenly changes. It is also possible to cache the resource representation initially received from the framework in connection with the endpoint until the last chunk is requested. If the server implements caching of framework responses (and the application supports it by setting

CoAP option	HTTP header
Accept	Accept, Accept-Charset
Content-Format	Content-Type
ETag (Not necessarily reversible)	ETag
Proxy-Uri, Proxy-Scheme	Not applicable
Max-Age	Cache-Control: max-age
Location-Path, Location-Query	Location
If-Match	If-Match
If-None-Match	If-None-Match
Size1	Not existing
Size2 (if block-wise or 4.13)	Content-Length

Table 5.3.: Direct mappings from CoAP options to HTTP headers

caching headers), the performance can be increased, because the resource representation could be cached for all endpoints. Another approach would be using HTTP Range requests [RFC 7233] but that also would have to be supported manually by the application[17]. For now we intend to use the naive implementation, because it does not require developing effort on the Rack application side.

5.2.2. Content Negotiation and Transcoding

For the Content Negotiation to be translated between both protocols (see section 5.3 of [RFC 7231] for HTTP and section 5.5.4 of [RFC 7252] for CoAP), the accept option of a CoAP request has to be rewritten into a HTTP media type and a content encoding, which can be passed in to the Rack application through the `HTTP_ACCEPT` field in the environment. If possible, the application responds with an appropriate body representation and the correctly set `Content-Type` header. Section 6.2 of [core-http-mapping-06] further specifies the media type mapping and defines a *Loose Media Type Mapping* that can be used to generalize HTTP media types to ones supported by CoAP. In section 6.3 also an algorithmic conversion from HTTP Internet Media Type to CoAP Content Format is given.

For constrained nodes, a compact payload serialization format such as CBOR [RFC 7049] is essential. It is possible to transcode between payload content formats to make use of efficient formats (as mentioned in section 6.4 of [core-http-mapping-06]). JSON

[17]As of Rails 4.2: `curl -sI http://localhost:3000 | grep Accept-Ranges`

can be mapped transparently to CBOR. This is particularly useful, because JSON is easily generated out of a Rails 4 application by direct conversion of an object (with the `to_json` instance method) from the controller or a `jbuilder` template as view specification. To support formats that can not directly be converted to JSON, conversion methods and possibly a Domain Specific Language (DSL) for easier templating would be needed, therefore this work currently focuses solely on CBOR.

The cbor gem [1] is a ready-to-use library "based on the (polished) high-performance msgpack-ruby code" for serialization of Ruby objects into CBOR and vice versa. Although the gem currently does not support JRuby as a Ruby VM, we favor it over cbor-simple[18], because of its performance.

Transcoding of incoming requests with CBOR payloads to JSON is less straightforward. Other than JSON, CBOR does not only support Strings as keys in Hashes. When converting a Hash to JSON, the standard library `JSON.parse` method calls `#to_s` on the keys. An exemplary Hash `{1 => 2}` decoded from CBOR would become the JSON String `'{"1" => 2}'`. So if we actually want to pass the information that the key `1` is a Fixnum correctly into the controller, we can not use JSON. A method would have to be found how to convert incoming CBOR payload directly into the Rails `params` Hash as it is done with incoming JSON data by Rails. A less elegant way would be letting the application programmer parse the CBOR body in the controller. Handing down a Ruby object deserialized from CBOR in the Rack environment (for example as value of `coap.cbor`) would provide a little convenience and prevent redundant parsing.

When automatically translating JSON bodies to CBOR, protocol compatibility and transparency have to be preserved. If the client does not specify a content format preference through the accept option, a JSON response of the Rack application can be translated transparently into CBOR. The same applies if the client specifies CBOR as its preferred content format. If the client actually requests the body represented in a certain content format different from CBOR (e.g. JSON), the answer must also be in that content format to not break with Content Negotiation as required by CoAP or HTTP.

Transcoding could be implemented as a Rack middleware that is also a *Celluloid* actor. A message containing information about which Media Types for which controller action are to be transcoded could be sent from the Rails controller to the `Transcoding` middleware actor. This would enable the application developer to decide for which controller and actions a configurable transcoding happens. However, currently we know of no way to determine the controller and action in a middleware `call`. Without this

[18]https://github.com/lucas-clemente/cbor-simple

information, a transcoding middleware has no advantages over transcoding the body representation outside of the Rack middleware stack.

5.2.3. Chunked Transfer Coding

Since version 1.5.0, Rack supports HTTP/1.1 Chunked Transfer Coding (see section 4.1 of [RFC 7230]). In a plain Rack application as seen in "Minimal Rack application" (Listing 3.1) on page 11 for example, the response body is chunked transparently if the *Content-Length* header is omitted. There is not really a CoAP extension which is comparable, although with [core-coap-streaming-00] there is an expired draft. This, however, needs the client to explicitly send an observe register (see sections 2.1 of [core-coap-streaming-00] and 3.1 of [core-observe-16]). The server has to reassemble the chunked resource representation, possibly cache it and return it to the client.

5.2.4. Resource Discovery

The CoAP Resource Discovery (section 7 of [RFC 7252]) is especially useful in M2M communication. After nodes are discovered through Service Discovery (see section 7.1), the `.well-known/core` interface according to [RFC 6690] (as referenced by section 7.2) can be used to discover a node's resources and their attributes such as the content format or an interface description. An automatic listing of a rack applications resources is examined in the next subsection. The Multicast subsection describes the requirements of multicast enabled resource discovery. The design of a Resource Directory (RD) [core-resource-directory-02], which maintains CoRE Link collections of different nodes, is considered in the Resource Directory subsection.

Reflection of Routing in .well-known/core

The `.well-known/core` resource returning a description of the available resources of the Rack application in the CoRE Link Format [RFC 6690] can be provided by the CoAP server as a Rack middleware. By including the server into a Rails project, a middleware can be inserted into the stack automatically utilizing a Railtie [28]. There is no general way to list the resources provided by a Rack application or an application written in any Rack based web framework. Using Rails, an enumeration of the defined routes is possible by processing `Rails.application.routes`. To support different Rack based web frameworks, it is necessary to create framework specific modules for the enumeration of resources.

The only attribute of a Link that can be determined by just enumerating the routes is `href`. A method of annotating resources with information about Resource Type (rt), Interface Description (if), Maximum Size Estimate (sz), and the other possible attributes for CoRE Link descriptions has to be developed. One possibility would be providing a method in the context of a controller defining both default attributes and attributes for specific resources. Listing 5.5 provides an example of how this could be used. On call of the `discoverable` method, these attributes have to be communicated to the instance of the Resource Discovery Rack middleware, which is instantiated at the start of the server. Besides dynamically changing a method of the middleware instance to return the right structure containing any defined link attributes, the middleware instance could also be a *Celluloid* actor that can receive messages triggered by the call of `discoverable` in the controller context. An overview in the form of a sequence diagram is to be seen in Figure 5.2. Possible performance impacts of *Celluloid* actor inclusion in the middleware stack have to be analyzed.

Rails returns the available resources as URI patterns. Those patterns are known for example from the output of the command `rake routes` ran in a Rails project. An URI pattern like `/things/:id` could be machine processable, if all involved endpoints support `:id` is a variable substitution. But this pattern follows no known standard. The URI Template standard [RFC 6570] defines ways of variable expansions for URIs. By this definition, the previously mentioned pattern could be represented by the template `/things/{id}`. This substitution can easily be performed by the middleware. However, URI Templates are not supported in CoRE Link URI-references.

Actors included in the middleware stack should be supervised and therefore restarted on crash, because otherwise any request fails after one actor in the stack died due to the impossibility of invoking the `call` method on a dead actor. This is not trivial, because the *Celluloid* API only provides a supervision possibility if the actor is instantiated in a special way but Rack expects to instanciate middleware objects itself. Probably this can be solved through a proxy object.

Multicast

The `.well-known/core` resource has to be offered additionally on multicast requests to the addresses defined in sections 8 and 12.8 of [RFC 7252] and section 2.2 of [RFC 7390]. The implementation guidance document draft [lwig-coap-01] does not state anything regarding the implementation of multicast Resource Discovery. To limit the size and number of answers, it is beneficial to support filtering the result set by the entries attributes. If the result set is empty, no response should be sent on multicast requests

Listing 5.5: Action annotation for Resource Discovery

```
class ThingsController < ApplicationController
  discoverable \
    default: { if: 'urn:things', ct: 'application/cbor' },
    index:   { if: 'urn:index' }

  def show
    render json: Thing.find(params[:id])
  end

  def index
    render json: Thing.all
  end
end
```

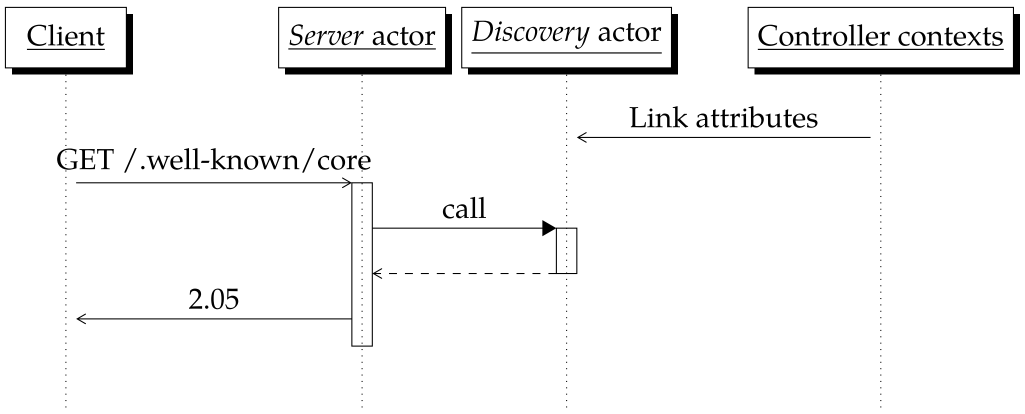

Figure 5.2.: Interaction regarding action annotation for Resource Discovery. The *Discovery* actor receives link attributes of resources from the respective controller. The resource descriptions returned on client request then contain these attributes.

(as it is described by section 8.2 of [RFC 7252] and section 4.1 of [RFC 6690]. Through a CoAP specific Rack environment entry (`coap.multicast`), it would be possible for applications to determine if an incoming request was received on a multicast address.

It has to be evaluated how the synchronous nature of HTTP and therefore Rack and Rails can be integrated for example with the leisure period requirement mentioned in section 8.2 of [RFC 7252].

Resource Directory (RD)

A Resource Directory [core-resource-directory-02] "hosts descriptions of resources held on other servers, allowing lookups to be performed for those resources". The specification can be implemented as a Rails application or mountable engine [19]. The provision of the `.well-known/core` resource from a Rack middleware (as specified earlier in this section) will not be used in this case, because it is integrated with the routing and controllers of the Rails application. As the default content format is `application/link-format`, we need a way to make it configurable for the application.

5.2.5. Observe

The CGI inspired design of Rack aimed at synchronous communications complicates the integration of asynchronous concepts like *WebSocket* or – in our case – CoAP observe [core-observe-16]. *Rack Hijack* [Rack] describes the proceeding of handing over the socket object from the server to the application (controllers) as a Rack environment option. For *WebSockets*, there is a gem called *tubesock*[19] utilizing *Rack Hijack*, which could be used as an inspiration for a gem that provides support for *Observe* in Rails in a similar manner. However, this is rather bad design, because placing the observe logic out of the server and into an extra gem used from the framework facilitates code duplication and breaks the model of a layered stack of abstractions from a object-oriented design point of view. In *tubesock*, this may even be a valid design choice, because the *WebSocket* protocol [RFC 6455] is out-of-band and not handled by the main web server. The code to be executed is handed over to *tubesock* by callbacks, which conflicts with the declarative, object-oriented design philosophy of *Celluloid*. This method also circumvents the Rack middleware stack.

Another approach would be viewing a component handling observe notifications and observer management as an actor (realized with *Celluloid*). On incoming observe reg-

[19] `https://github.com/ngauthier/tubesock`

ister or deregister requests, the server actor(s) message the observe actor. The observe actor maintains the list of registered observers and periodically determines whether to send observe notifications (further called *Observe tick*). The interaction of actors among each other and with the client and the Rack application in the cases register, deregister and tick are visualized in Figure 5.1. The most quick and easy implementation for determining whether an update is necessary, is to clone the original request of a resource, pass it to the framework (Rack application) and send a notification if the resource changed. With the framework answer's validator header fields `Last-Modified` and `ETag` (see sections 2.2 and 2.3 of [RFC 7232]) (or, if they are missing, a hash sum of the body) it can be determined if the framework returned a changed body and thus a client has to be notified about the updated resource. An optimization for frameworks supporting validator header fields is to facilitate conditional requests. This way it is possible, the overhead of regenerating the resource representation can be stinted. For an exemplary interaction in this case, refer to Figure 5.3. Rails uses *Rack::ETag* for entity tags which is a middleware that can also be used to provide plain Rack applications or other Rack based frameworks with `ETag` support. In Rails, the decision whether the resource representation has to be rendered is made from controller code and thereby explicitly necessary. The notification can be omitted if the entity tag did not change and the max-age time of the last notification is not expired. If this max-age time is expired and the entity tag did not change, a notification with CoAP *2.03 Valid* response code is sent.

For the observer list management, besides adding observers to resources and deleting them, the observe actor has to implement a "garbage collection" for obsolete observe relationships (as described in section 3.6 of [core-observe-16]). If observe notifications are always sent as non-confirmable messages and the server does not send any other confirmable messages, the retransmission logic can be omitted. This saves some memory and CPU resources. The specification explicitly states, "A notification can be confirmable or non-confirmable, i.e., it can be sent in a confirmable or a non-confirmable message. The message type used for a notification is independent of the type used for the request and of any previous notification." (see section 3.5 of [core-observe-16]).

For the application code it would be possible to utilize the HTTP/1.1 caching headers like `Expires` or `Cache-Control max-age` (see sections 5.3 and 5.2 of [RFC 7234]) to communicate when a resource should be polled again and in which interval a resource change should be polled for. Polling for resource updates wastes resources, so for a more performant solution, the framework or application code would have to message the observe actor about a resource update. If the resource is a representation of a list of saved entities for example, the framework would have to notify the observe actor on

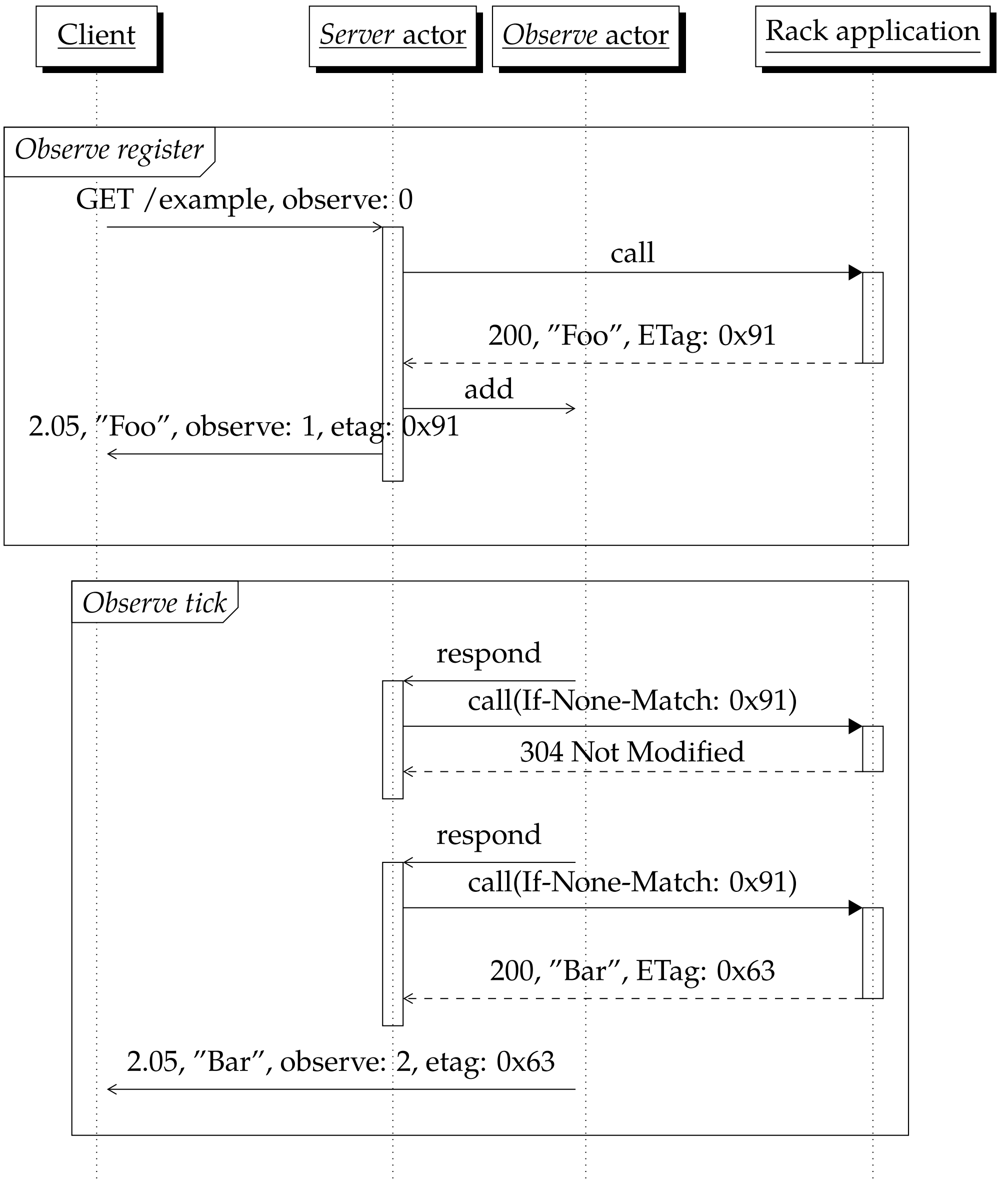

Figure 5.3.: Observe interaction with conditional requests. On *Observe register* the `ETag` value of the Rack response is saved and later used to determine if the resource representation changed and an update is necessary.

Listing 5.6: Caching options in Rails controller

```ruby
class ThingsController < ApplicationController
  def show
    @thing = Thing.find(params[:id])

    # Set Cache-Control max-age to 60 seconds.
    # Manipulates observe actor polling frequency.
    expires_in 1.minutes

    # Set Last-Modified and ETag headers
    # and respond only if conditional request matches.
    render json: @thing if stale? @thing
  end
end
```

each creation or deletion of an entity. However, this problem can not generally be solved by callbacks triggered on database changes. Explicitly signalling a resource update with a certain method call from the Rails controller is not possible, since the controller code is only called upon request from the server or observe actors and not running as an actor itself.

In Listing 5.6 a Rails controller code example is shown that both manipulates the polling frequency of the observe actor for this resource and only renders the resource representation if a conditional request by the observe actor indicates a change. *Rails::Observers*[20] can be used to further control cache invalidation.

[20]https://github.com/rails/rails-observers

Chapter 6

Implementation

This chapter documents the implementation drafted in the previous chapter. Each component realized as an independent module is explained in detail in the following sections. Table 6.1 gives an overview of the components we worked on, their respective licenses, and the sections covering their implementation.

Component	License	Section
coap (Library)	MIT	6.2
david (Server)	GPL	6.3
core-rd (RD)	AGPL	6.5

Table 6.1.: Implemented Components

6.1. External Libraries

For all implemented components, we make use of several external libraries. The most important ones are listed in Table 6.2. A more exhaustive list can be obtained from the `Gemfile.lock` files inside the project repositories.

Library	Version	License
cbor	0.5.9	Apache 2.0
celluloid	0.16	MIT
celluloid-io	0.16.2	MIT
rack	1.6	MIT
rails	4.2	MIT
rspec	3.2	MIT

Table 6.2.: External Libraries

A small change was integrated into the *Celluloid::IO* project. Through inclusion of *Celluloid::IO*, socket classes are wrapped in a way that prevents blocking usage. Methods which do not block are directly delegated to the wrapped standard library socket class. A delegation for the `addr` instance method was added in the course of this work[1].

6.2. CoAP Library

The CoAP library based on the message parser by Carsten Bormann published by the GOBI project[2] was forked on GitHub[3]. The original project is released under the terms of the MIT license[4], and so is the fork we developed. One of the original authors, Simon Frerichs, gave us access to the coap gem on rubygems.org and the first version of our fork we released was 0.1.0[5]. The current release version is 0.1.1. Initial objectives were: Restructurization, modularization (for code reuse in the server component), removal of code duplication, freeing of tests from external services, and bug fixes. Another important objective was to make any socket communications non-blocking and shared data structures compatible with concurrent access. The following lists extracted from the commit log summarize the individual changes to the CoAP library[6] by category. The whole commit log in comparison to the original source code can be found on GitHub[7].

Refactoring

- Refactoring and simplification of tests
- Refactoring of socket abstraction class
- Modularization of message parser
- Refactoring of central client method (especially method extraction)

Janitor Tasks

- Update message parser
- CoRE namespace
- More current Ruby version on Travis CI
- Gem updates

[1] `https://github.com/celluloid/celluloid-io/pull/114`
[2] `https://github.com/SmallLars/coap`
[3] `https://github.com/nning/coap`
[4] `https://github.com/SmallLars/coap/blob/master/LICENSE`
[5] `https://rubygems.org/gems/coap`
[6] `https://github.com/nning/coap/commits/master`
[7] `https://github.com/nning/coap/compare/SmallLars:master...master`

Code Optimization and Reimplementation

- Reimplementation of command line utility
- Unification and better documentation of client method arguments
- Bugfixes in message parser (path and query encoding)
- `CoAP::Message#to_s`
- Preparations for JRuby compatibility
- Content format registry
- CoRE Link format [RFC 6690] implementation
- Changed client method argument order from {host, port, path, method, payload, options, callback} to {method, path, host, port, payload, options, callback} (all but method and path are omitable)
- Changed socket abstraction to non-blocking transmission layer class (`Transmission`) based on `Celluloid::IO`
- Separate support and ACK on CON in `Transmission` class
- JRuby support
- Optimized chunkifying payload for block-wise requests
- Corrected maximum random numbers for mid and token
- Configurable socket for `Transmission`
- CoAP transmission Finite-state machines (FSMs) (not yet properly integrated)
- Reimplementation of `Block` class
- Utility code to determine Operating System, check if network interface is up and network interface name to index conversion (needed in context of server multicast implementation)
- Support for setting the query in client

Testing

- Preparations for RSpec [30]
- Porting of client tests to RSpec (block1, observe, and separate)

6.3. David

The core of this work – the CoAP server component with a Rack interface – is called *David* in reference to Goliath (an asynchronous HTTP server with a Rack interface). It is realized as a Ruby gem[8] and can be used as a drop-in replacement for HTTP servers in a Rails application. It is released under the terms of the General Public License (GPL)

[8] https://rubygems.org/gems/david

[GPLv3] and downloadable on GitHub[9]. James Fairbairn originally held *david* as a name on rubygems.org but kindly gave it to us. The first public release was version 0.3.0.pre. The current stable release is 0.4.3 and all code descriptions in this section refer to that version.

The code was developed mostly on MRI versions between 2.1.4 and 2.2.0 but it was also optimized for and tested on MRI 1.9.3-p551 and 2.3.0-dev, JRuby 1.7.18 and 9.0.0.0-pre, and Rubinius from 2.4.0 till 2.5.2. In JRuby and Rubinius, the detection of multicast messages does not work because of the platforms' `Socket` APIs (for further details, see subsection 6.3.3). The message handling of David running on Rubinius suffers from a deadlock problem that could not been solved yet. Some of the server's features such as application Resource Discovery (see section 6.4) are specific or at least tied to Rails and do not work with every Rack based framework.

The following subsections handle specific aspects of the implementation.

6.3.1. Options

Besides configuration through the commandline (e.g. with the `-O` option of `rackup`), the server's options can be set from the Rails application config (e.g. in `config/application.rb`. The possible configuration options are listed in Table 6.3.

[9] `https://github.com/nning/david`

Rack key	Rails key	Default	Semantics
Block	coap.block	true	Block-wise transfer support
CBOR	coap.cbor	false	JSON/CBOR transcoding
DefaultFormat	coap.default_format		Default Content-Type (if CoAP accept omitted)
Host		::1 / ::	Server listening host
Log		info	Log level (none or debug)
MinimalMapping		false	Minimal mapping (see section 7.2.2)
Multicast	coap.multicast	true	Multicast support
Observe	coap.observe	true	Observe support
	coap.only	true	Removes HTTP middleware and adds David as default Rack handler
Port		5683	Server listening port
	coap.resource_discovery	true	Provision of `.well-known/core`

Table 6.3.: David configuration options

6.3.2. Rack

David adds some application specific entries to the Rack environment which are listed in Table 6.4.

Key	Value class	Semantics
coap.version	Integer	Protocol version of CoAP request
coap.multicast	Boolean	Marks whether request was received via multicast
coap.dtls	String	DTLS mode (as defined in section 9 of [RFC 7252])
coap.dtls.id	String	DTLS identity
coap.cbor	Object	Ruby object deserialized from CBOR

Table 6.4.: Rack environment additions of David

We monkey-patched the Rack code to use David as the default Rack handler unless Rails is loaded and `config.coap.only` is set to `false` (see `lib/david/guerilla/rack/handler.rb`). This way executing `rackup` or `rails s` starts the CoAP server, and does not try the original Rack handler order (thin, puma, webrick). The actual Rack handler is defined in lib/david/rack/handler/david.rb and performs the startup of the different server actors (Server, GarbageCollector, and Observe) in a *Celluloid* supervision group.

Automatic provision of Resource Discovery trough a .well-known/core resource and return of exceptions caused by the application in a human readable format are created as Rack middleware classes David::ResourceDiscovery (see section 6.4) and David::ShowExceptions (see lib/david/show_exceptions.rb). They are inserted into the middleware stack by the David::Railties::Middleware railtie (see lib/david/railties/middleware.rb). Other middleware shipped with Rails is removed if it (currently) provides no features in connection with CoAP (see line 8 ff.). It is possible to keep it included in the middleware stack by setting coap.only to false in the Rails application config.

6.3.3. Protocol Implementation

Parsing incoming transmissions does not utilize other shared functionality from the CoAP library than the CoAP::Message.parse method, which was provided by Carsten Bormann and only changed minimally. For answering or initiating transmissions, the CoAP::Transmission class (see lib/core/coap/transmission.rb in version 0.1.1 of the coap gem source) was used first to avoid code duplication. It abstracts timeouts, retransmissions, ACK answering, separate answers and checking of message identifiers and tokens. For more information, refer to section 6.2. However, when optimizing the server for throughput performance it became obvious that this shared functionality also introduced overhead. Most features were too client centric anyway: Retransmissions, incoming separate transmissions and message token management are not necessarily useful for a server. These shared code parts are also not modular enough to allow a single place for incoming message dispatching. So after a redesign of the architecture, messages are sent by direct invocation of send on the server socket. References to that socket are passed between actors. The first version with the revised architecture is 0.4.0. A cache for message correlation and deduplication of confirmable requests is currently solved through a Ruby hash with endpoint and message identifier as key and the cached response and a timestamp for cache invalidation as values (see lib/david/server/mid_cache.rb). The GarbageCollector actor (see lib/david/garbage_collector.rb) periodically cleans the cache from obsolete messages. Using the reimplemented block support of the coap gem, block-wise transfers [core-block-17] were integrated into the server. However, the current state only supports block-wise responses; block-wise transfers for assembling requests are not supported. Separate responses as designed in section 5.1.2 were not implemented yet as they make a bigger change to the current request/response cycle necessary.

Observe support is accomplished through a *Celluloid* actor handling the notifications. The `Server` actor adds or removes listeners via `#add` or `#delete` messages to the `Observe` actor (as depicted in Figure 5.3; also see line 133 of `lib/david/server/respond.rb`). Listeners are saved in a Ruby hash with endpoint and message token as key and observe number (initially 0), original request, rack environment, originally returned `ETag`, and a timestamp for cache invalidation as values (see `lib/david/observe.rb`). In a static interval of 3 seconds by default, the `Observe` actor checks every target resource of the observe relationships on changes and sends updates to the endpoint if necessary. The update check is solved by requesting a full framework answer for the original request from the `Server` actor. If the `ETag` differs between the original request and response, an observe notification is sent to the endpoint. The observe relationship persists unless the endpoint answers with a reset message to a notification, explicitly deregisters or the update check is answered with an error response code by the framework or application. There is potential for performance improvements in the current implementation, because the `Observe` actor neither refreshes every individual resource only once per tick nor makes effective use of HTTP caching as drafted in subsection 5.2.5. A correctly set `ETag` header is mandatory for the `Observe` actor to decide if an observe notification has to be sent. Rails automatically includes `Rack::ETag`, which is a Rack middleware that transparently adds an `ETag` header. This middleware can be included manually when using more minimal frameworks. As an example, see line 6 of `config.ru`, a Rack configuration that starts a plain Rack application for testing purposes.

Listing 6.1 shows the implementation of the event loop. Inside a class including `Celluloid::IO`, `recvfrom` automatically uses the `Celluloid::IO` event loop. There were some changes necessary to the original message handling to determine if a request has been received on a multicast address. The non-blocking `#recvfrom` method exposed by `Celluloid::IO::UDPSocket` class instances does not return sufficient information. Therefore `#recvmsg_nonblock` of the wrapped `UDPSocket` class instance was used to receive data and information on the target address from the socket. The event loop was implemented as in `#recvfrom` of `Celluloid::IO::UDPSocket` (see the `Celluloid::IO::UDPSocket` implementation and line 45 of `lib/david/server.rb` for reference) using `Celluloid::IO.wait_readable` which is based on the OS independent I/O se-

Listing 6.1: Server Event Loop

```
1  loop do
2    if jruby_or_rbx?
3      dispatch(*@socket.recvfrom(1152))
4    else
5      begin
6        dispatch(*@socket.to_io.recvmsg_nonblock)
7      rescue ::IO::WaitReadable
8        Celluloid::IO.wait_readable(@socket)
9        retry
10     end
11   end
12 end
```

lector API of nio4r[10]. In JRuby and Rubinius we use the non-blocking `recvfrom` method provided by *Celluloid::IO*. This change was not proposed to the *Celluloid::IO* project, because we could not get a Ruby VM independent implementation working due to the absence of a `recvmsg_nonblock` method in JRuby and Rubinius. Since *Celluloid::IO* aims on VM independence, this effort was canceled for now.

6.3.4. Concurrency and Performance

For maximizing the packet handling throughput, it is important how concurrency is accomplished on packet receiving. The main code regarding concurrent handling of network packets resides in `lib/david/server.rb`. The event loop (line 41 ff. of that file) is also shown in Listing 6.1. The Rack handler starts the basic server actors `Server`, `Observe`, and `GarbageCollector`. Each actor runs in its own Thread and is supervised by *Celluloid* (restarted on errors). Afterwards the Rack handler code calls the `run` method on the `Server` actor blockingly (see line 15 of `lib/rack/handler/david.rb`).

The event loop receives incoming package data in a non-blocking way and calls the `dispatch` method for input handling (lines 3 and 6 of Listing 6.1). If the `IO` object is not readable, `Celluloid::IO` is used to select the next readable one (line 8). Benchmarks showed a significant performance increase through avoidance of calling `dispatch` asynchronously in a Fiber (by the `async` method provided by *Celluloid*). It is important that blocking I/O operations are not used by the framework or application

[10] `https://github.com/celluloid/nio4r`

programmer, because the generation of the Rack application's response happens inside the event loop and it stops other incoming messages from being processed. Non blocking database adapters have to be used in the Rack application. Support for separate answers would especially be useful in this context.

6.3.5. Protocol Translations

The CoAP methods are mapped directly to their HTTP counterparts and passed to the Rack compatible framework via the `REQUEST_METHOD` Rack environment entry. Messages with CoAP codes that are undefined method codes (see section 12.1.1 of [RFC 7252]) are dropped directly after parsing the message.

Most direct mappings of HTTP response codes and headers to CoAP response codes and options are performed as specified in section 5.2 through the methods defined in `lib/david/server/mapping.rb`. The methods are included into the `David::Server` class and used especially in its `respond` method defined in `lib/david/server/respond.rb`. The HTTP response codes returned in Rack responses are mapped according to the procedures described in section 5.2. In the Ruby code the map is represented through the constant `HTTP_TO_COAP_CODES`. With the Rack option *MinimalMapping*, the default mapping of HTTP to CoAP response codes can be deactivated in case application programmers want to explicitly return a certain code but can not use a Float (see section 7.2.2). Currently the implementation does not map HTTP return code **and** methods to CoAP codes. Also the translation of HTTP redirects to JSON redirect descriptions is not implemented. Both were queued up because of the time constraints of this work.

The `Mapping` module also contains methods to convert certain headers from HTTP to CoAP or the other way round. `etag_to_coap` converts the HTTP ETag header value string (containing a hash of the body) by interpreting its first bytes (defaulting to four) as an integer. `location_to_coap` splits the Location header string value into CoAP Location-Path segments. `max_age_to_coap` returns the Cache-Control header's max-age value as an integer. The method `accept_to_http` is used to translate the Accept header from CoAP to HTTP and is therefore enabling Content Negotiation (as specified in subsection 5.2.2). The options Content-Format, Max-Age, Location-Path, and Size2 are mapped without individual methods of the `Mapping` module. The other direct header mappings drafted in Table 5.3 (Proxy-Uri, Proxy-Scheme, Location-Query, If-Match, If-None-Match, and Size1) are not implemented, yet.

Content transcoding of CBOR to JSON and vice versa is activated if the Rack environment option `CBOR` or the Rails option `config.coap.cbor` are set to `true`. It is deactivated by default. If the transcoding is activated and an incoming request has the content format *application/cbor* (60), the message body is parsed by the Ruby cbor gem [1] as CBOR into a Ruby object (see line 31 of `lib/david/server/respond.rb` and line 41 of `lib/david/server/mapping.rb`). This object is saved in the Rack environment (as value of the key `coap.cbor`) and then converted to JSON, wrapped in a StringIO instance and also handed down the Rack stack as Rack environment option `rack.input`. The Rack environment option `CONTENT_TYPE` is set to *application/json* if CBOR parsing and JSON serialization did not throw any exceptions. The `CONTENT_LENGTH` is updated according to the length of the resource represented as JSON. Rails integrates the deserialized object into the params Hash which is accessable in controller code. After a Rack response is obtained from the application, the response body is transcoded to CBOR if its content type is *application/json*. The resource representation is parsed as JSON into a Ruby object and then serialized into CBOR (see line 50 of `lib/david/server/respond.rb` and line 38 of `lib/david/server/mapping.rb`).

6.3.6. Architecture

We made especially use of *Celluloid* actors and the inclusion of modules into classes for modularization. Figure 6.1 and 6.2 show the object relations. The former concentrates on the actors mainly involved in request handling and response processing, whereas the latter shows Rack middleware and Rails extensions. In both graphs names are suffixed with a letter in square brackets that indicates the type. *C* stands for a Ruby class, *M* for a model, and *A* for a *Celluloid* actor.

6.4. Resource Discovery

The provision of the `.well-known/core` interface for resource discovery on an application implemented in Rails is solved through a Rack middleware that is also a *Celluloid* actor. The middleware code mainly resides in `lib/david/resource_discovery.rb`. Currently, it is Rails specific, so applications written with other Rack supporting frameworks would have to implement Resource Discovery by their own. On invocation of `call`, the actor responds with a cached list of resources in the CoRE Link Format [RFC 6690] if the request is a GET on `.well-known/core`. The responded resources are automatically detected via Rails routes and can be extended with link attributes by annotations in the controllers as depicted in Listing 5.5 and Figure 5.2. The annota-

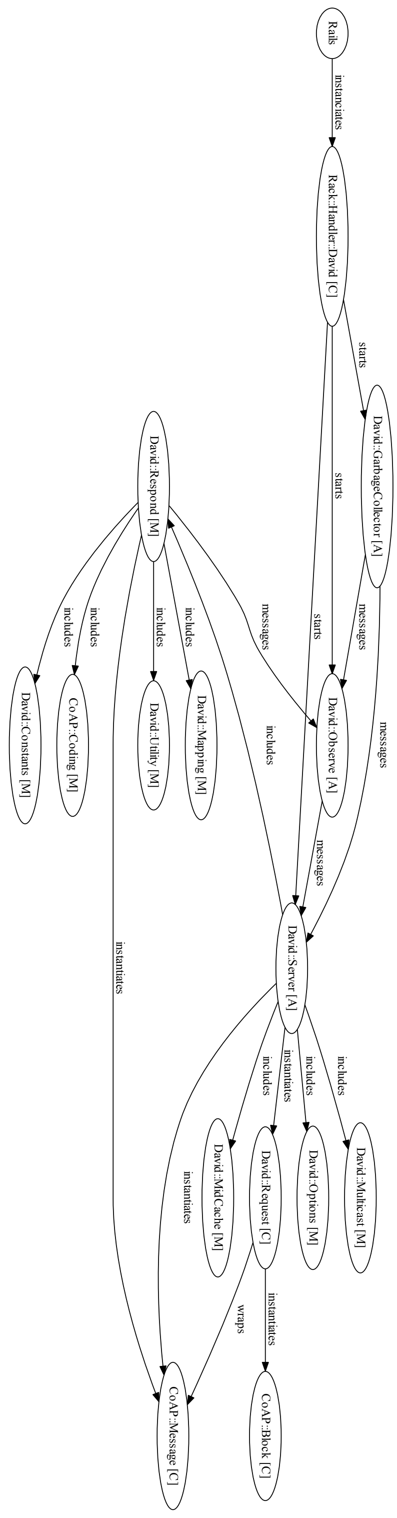

Figure 6.1.: Object Relations (Server)

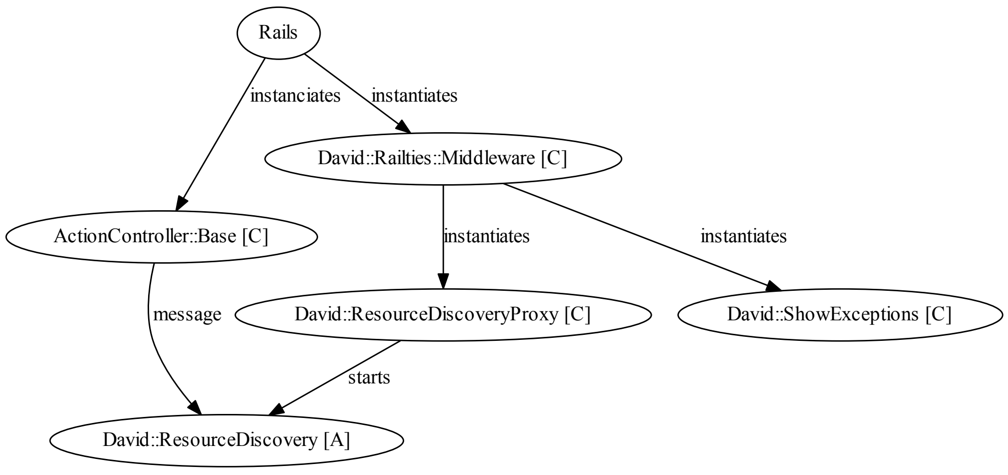
Rails
instanciates
instantiates
David::Railties::Middleware [C]
ActionController::Base [C]
instantiates
instantiates
message
David::ResourceDiscoveryProxy [C]
David::ShowExceptions [C]
starts
David::ResourceDiscovery [A]

Figure 6.2.: Object Relations (Middleware)

tion and registration of attributes at the Resource Discovery actor is accomplished by a class method named `discovery` defined on `ActiveController::Base` in `lib/david/rails/action_controller/base.rb`. Due to Rails lazy loading controller classes in the *development* environment, the `discoverable` class method is not called until the first time a route mapped to an action of that controller is requested. To overcome this limitation, link attributes for discovery would have to be specified together with the application routing.

Especially for Service and Resource Discovery, the basics of [RFC 7390] have been implemented in both David and the coap gem. The initialization of the socket is packaged in the `Multicast` module (see `lib/david/server/multicast.rb`). The multicast groups `ff02::1`, `ff02::fd`, `ff05::fd`, and `224.0.1.187` are joined by default (also see section 12.8 of [RFC 7252]). Some workarounds had to be implemented specific to OS X. The OS X kernel does not support joining a IPv6 multicast group on the default interface if the interface index is set to 0 (despite both the documentation and POSIX specification states otherwise), for example[11]. Utility methods for conversion of a network interface name to index and to test if an interface is activated have been integrated into the coap gem as extensions of the Ruby standard library `Socket` class (see `lib/core/core_ext/socket.rb`).

Other actors like `Server`, `Observe`, or `GarbageCollector` are instantiated

[11]https://lists.apple.com/archives/darwin-kernel/2014/Mar/msg00012.html

and supervised named by calling `supervise_as` instead of `new` (in lines 7-12 of `lib/rack/handler/david.rb`). However, when a middleware is inserted into the stack with a Railtie, not an instance of the class but the class itself is passed. This way a discovery actor could not be supervised by *Celluloid* and communicated with by name. Therefore, it has to register "manually" on initialization. This unfortunately does not mean it is also supervised. In this case, the solution is to proxy the actual Resource Discovery actor with a class behaving like an ordinary Rack middleware (as seen in `lib/david/resource_discovery_proxy.rb`). On initialization of this class, the Resource Discovery actor is instantiated and supervised named. When the `call` method is invoked, it is passed on to the actor.

6.5. Resource Directory (RD)

The Resource Directory component is realized as a Rails application (downloadable from GitHub[12]) in order to further test *David* and other implemented components regarding CoAP application development. The Rack middleware for Resource Discovery shipped with David is not used in this case, because the Rails application should provide a custom logic behind the `.well-known/core` resource and the middleware is tightly integrated with the routing and controllers. By default the CoAP server sets the `Accept` header to `application/json` if it is not specified by the client. In the RD application, the default content format is configured to `application/link-format`. Currently only the *RD Function Set* (see section 5 of [core-resource-directory-02]) is implemented completely. The *RD Lookup Function Set* (see section 7) is only implemented in a way that allows for the lookup of resources (*res* lookup type). We did not implement the *Group Function Set* (see section 6) or the lookup types related to that set.

There are Active Record models for Resource Registrations, Typed Links, and their Target Attributes (see the `app/models` directory in the `core-rd` repository). Controllers exist per function set and for the `.well-known/core` ressource (see `app/controllers`). We developed the application utilizing TDD with RSpec.

[12]`https://github.com/nning/core-rd`

Chapter 7

Evaluation

To support the development process and to measure code quality and sustainability, unit tests were written. The test code coverage and the code climate are also recorded. Our considerations of the research question "How to handle a huge amount of IoT devices?" are evaluated with performance benchmarks. The interoperability of the different interfaces of the server were benchmarked to analyze our considerations regarding to the research questions covering the CoAP implementation, the Rack interface, and translations of headers and payloads between HTTP and CoAP.

7.1. Unit Tests, Code Coverage, and Code Climate

The CoAP library and client, the server (David), and the RD implementation (core-rd) were developed mostly test-driven with RSpec [30]. For the former some essential tests were ported to and new ones were written with RSpec. For the other two components RSpec was used exclusively. Although it offers some features supporting Behavior-Driven Development (BDD), we used RSpec only to write DRY specifications of the codes functionality. The Test-driven Development of the RD showed that RSpec can seamlessly be used for the development of CoAP applications with Rails.

The test coverage is determined by Coveralls, a web service that gets coverage measurement values from test runs in the Continuous Integration (CI) system using the simplecov gem[1]. The results are prepared for easy analyzing by developers and displayed publicly. We used CodeClimate – another similar service – for measuring for example the complexity of methods by counting conditional branches and method calls. Classes and modules are classified by their complexity into categories from A (best) until F (worst). A code climate score (between 0.0 and 4.0) is computed from that information that indicates code quality.

For the fork of the coap library gem, the code climate score was raised from 0.74 to

[1]https://rubygems.org/gems/simplecov

2.66[2] and the test coverage from 88.19% to 93.06%[3] (although the total count of code lines increased from about 1270 to 1900). The test coverage for core-rd – the RD draft [core-resource-directory-02] implemented in Rails – is about 90.01%[4], the CodeClimate score is 3.37[5]. The source code of David scores 3.57[6] on CodeClimate and has a test coverage of 92.94%[7].

7.2. Benchmarking

We benchmarked the performance and interoperability of the developed server software. The performance was tested by measuring the handled requests per second in different Ruby VMs and with Rack and Rails. For the interoperability, several tests were conducted per interface.

7.2.1. Performance

We tested the requests David handles (receives, parses, processes, and answers) per second with Rack and Rails applications simply returning a plain "Hello World!" string. This *requests per second* value is measured with a number of concurrent clients that increases from 10 to 10,000 (by 10 below 100, 100 below 1,000 and 1,000 below 10,000) simulated with *Cf-CoAPBench* 1.0.0-M3[8]. We use a server running on loopback instead of a physical network and the different Ruby VMs mentioned below. The exact versions are noted as output from the `ruby -v` command. Benchmarking for each concurrency value is performed for 30 seconds. The average value of three subsequent runs is recorded. After each benchmark, we wait 15 seconds. At the beginning of the testing of each VM or framework, a 60 seconds warmup is done. The benchmark system is a notebook with a Core i7-3520M CPU and 16 GiB of RAM running Arch Linux. Benchmarks were conducted on the core repository linux kernel version 3.18.6-1 without further tuning. We only increased the maximum number of open file descriptors to enable 100,000 open UDP sockets. The CPU supports Hyper-Threading and regularly runs at 2.9 GHz and 3.9 GHz with only one active core. Server processes are started with `rackup` and a framework specific rackup configu-

[2] https://codeclimate.com/github/nning/coap
[3] https://coveralls.io/r/nning/coap
[4] https://coveralls.io/r/nning/core-rd
[5] https://codeclimate.com/github/nning/core-rd
[6] https://codeclimate.com/github/nning/david
[7] https://coveralls.io/r/nning/david
[8] https://github.com/eclipse/californium.tools/tree/1.0.0-M3/cf-coapbench

Listing 7.1: Performance benchmark environment variables

```
export RUBY_GC_MALLOC_LIMIT=134217728
export RUBY_GC_HEAP_FREE_SLOTS=200000
export JRUBY_OPTS="--server"
```

ration file (`benchmarks/rackup/rack.ru` and `benchmarks/rackup/rails.ru`) and the benchmarking uses the script at `benchmarks/stress.sh`. The server release `0.4.1` is used. It was planned to also test Rubinius 2.5.2 as a Ruby VM but because of a deadlock issue that could not be fixed yet, this benchmark is postponed for now.

1. MRI 2.2.0p0
   (2014-12-25 revision 49005)

2. MRI 2.3.0dev
   (2015-02-12 trunk 49574)

3. JRuby 1.7.19
   (1.9.3p551) 2015-01-29 20786bd on OpenJDK 64-Bit Server VM 1.7.0_75-b13 +jit

4. JRuby 9.0.0.0-pre1
   (2.2.0p0) 2015-01-20 d537cab OpenJDK 64-Bit Server VM 24.75-b04 on 1.7.0_75-b13 +jit

The garbage collection of both MRI versions was tuned by setting environment variables to increase the memory allocation maximum (128M) and to prepare a certain amount of free slots after starting the VM (200000). JRuby has been configured to use the Server Java Virtual Machine (JVM). Listing 7.1 shows the environment variables and their values.

We benchmarked Reel [29] on the same hardware with 1,000 and 10,000 concurrent clients to get a reference value to a HTTP server based on a very similar concurrency model and also on *Celluloid* and *Celluloid::IO*. It achieved 3,391.6 requests per second (RPS) at 1,000 and 3,414.4 RPS at 10,000 concurrent connections running in MRI 2.2.0p0.

We also tried to perform a performance benchmark with David running on an *Ubiquiti EdgeMAX Lite* router, as it is quite customizable and comparable to common customer router hardware. However, we could not get the performance benchmark running even after extensive experiments. See section B.2 for installation instructions on that hardware and a documentation of the steps taken by us.

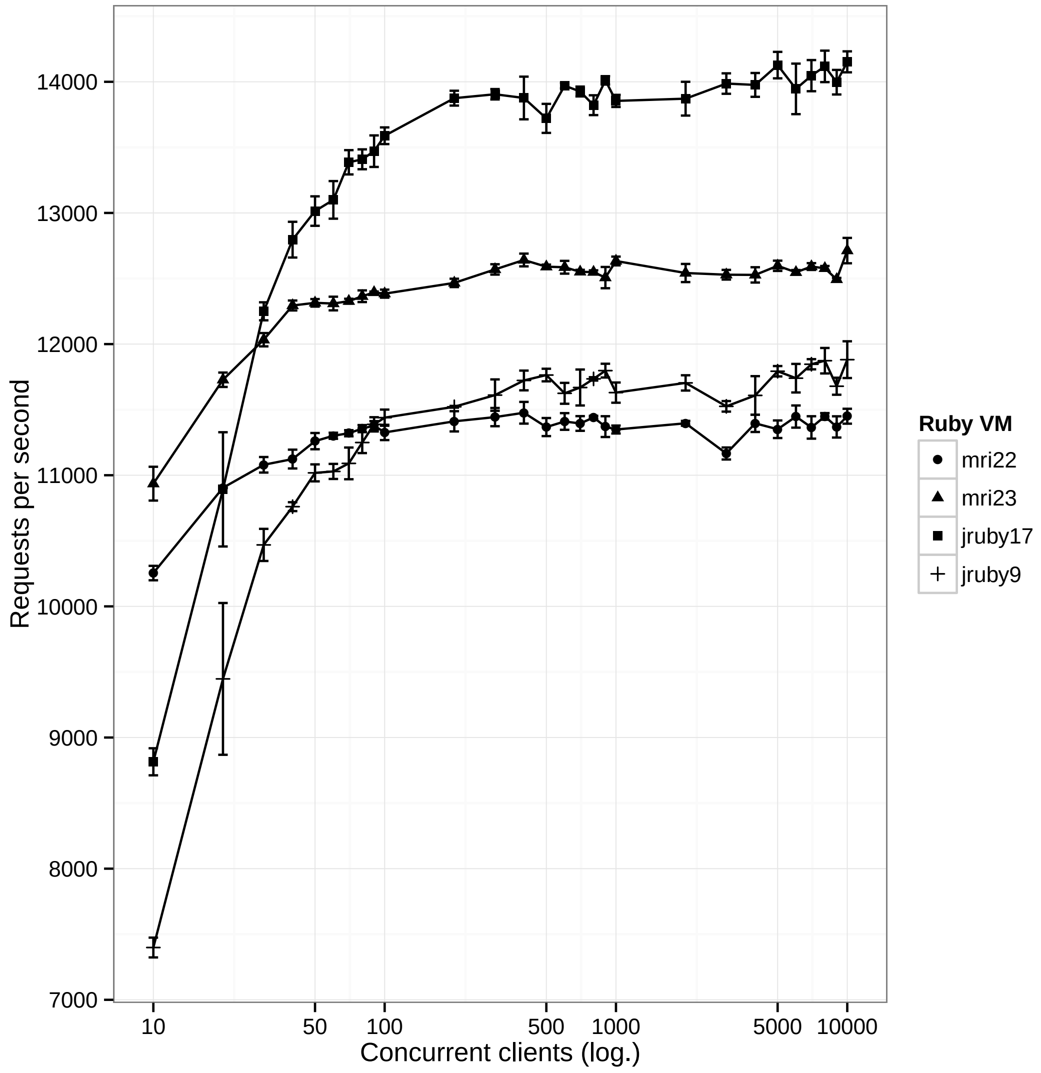

Figure 7.1.: Throughput by Ruby VM with Rack

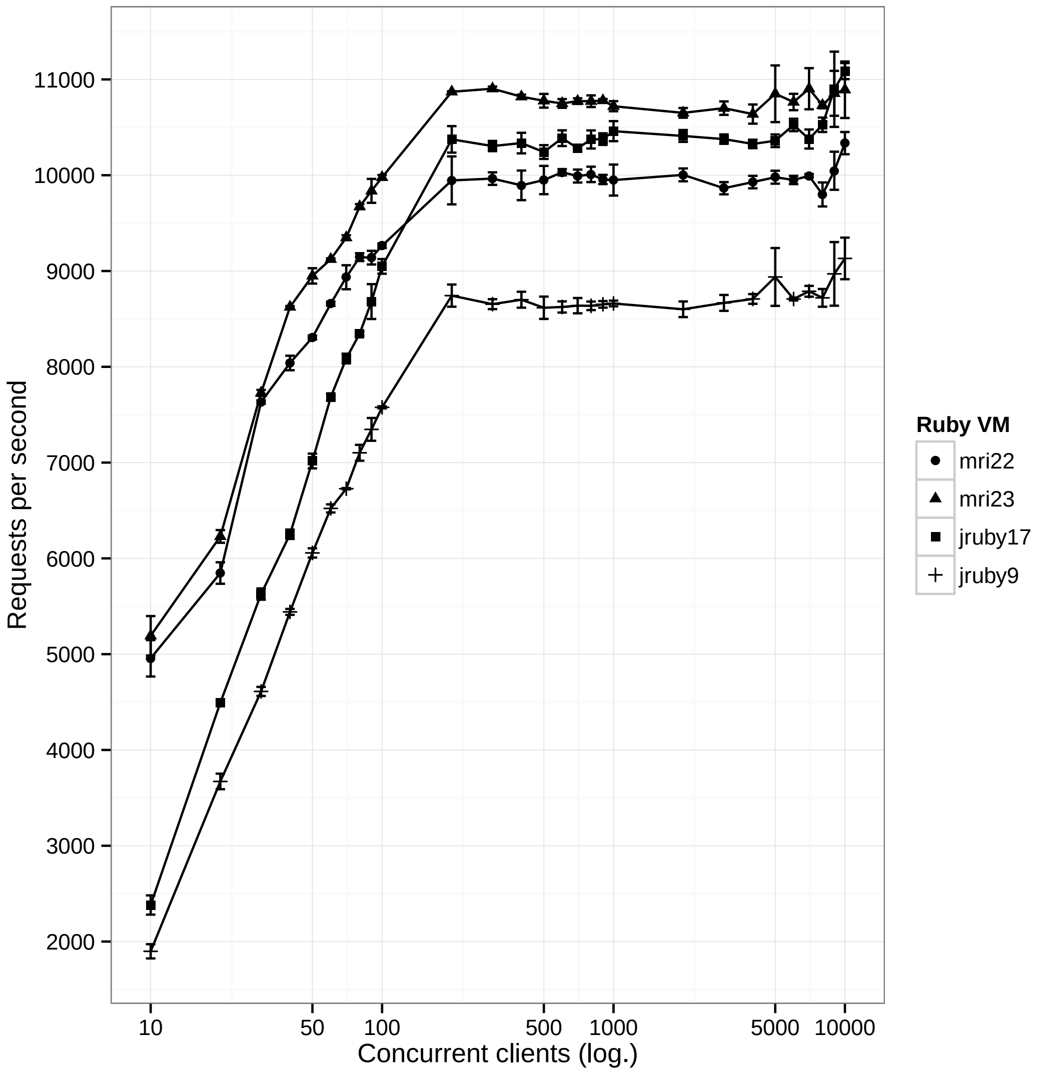

Figure 7.2.: Throughput by Ruby VM with Rails

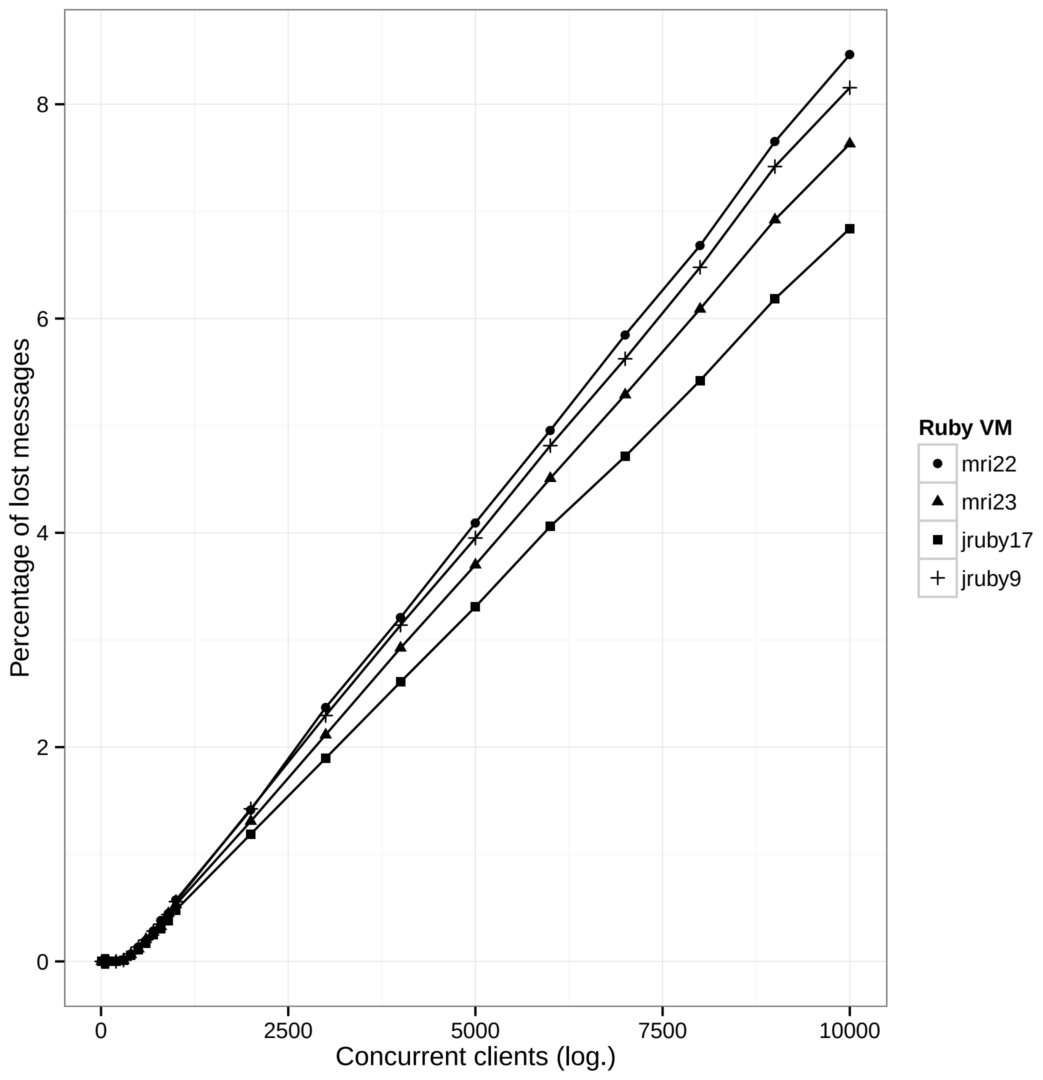

Figure 7.3.: Message timeouts by Ruby VM with Rack

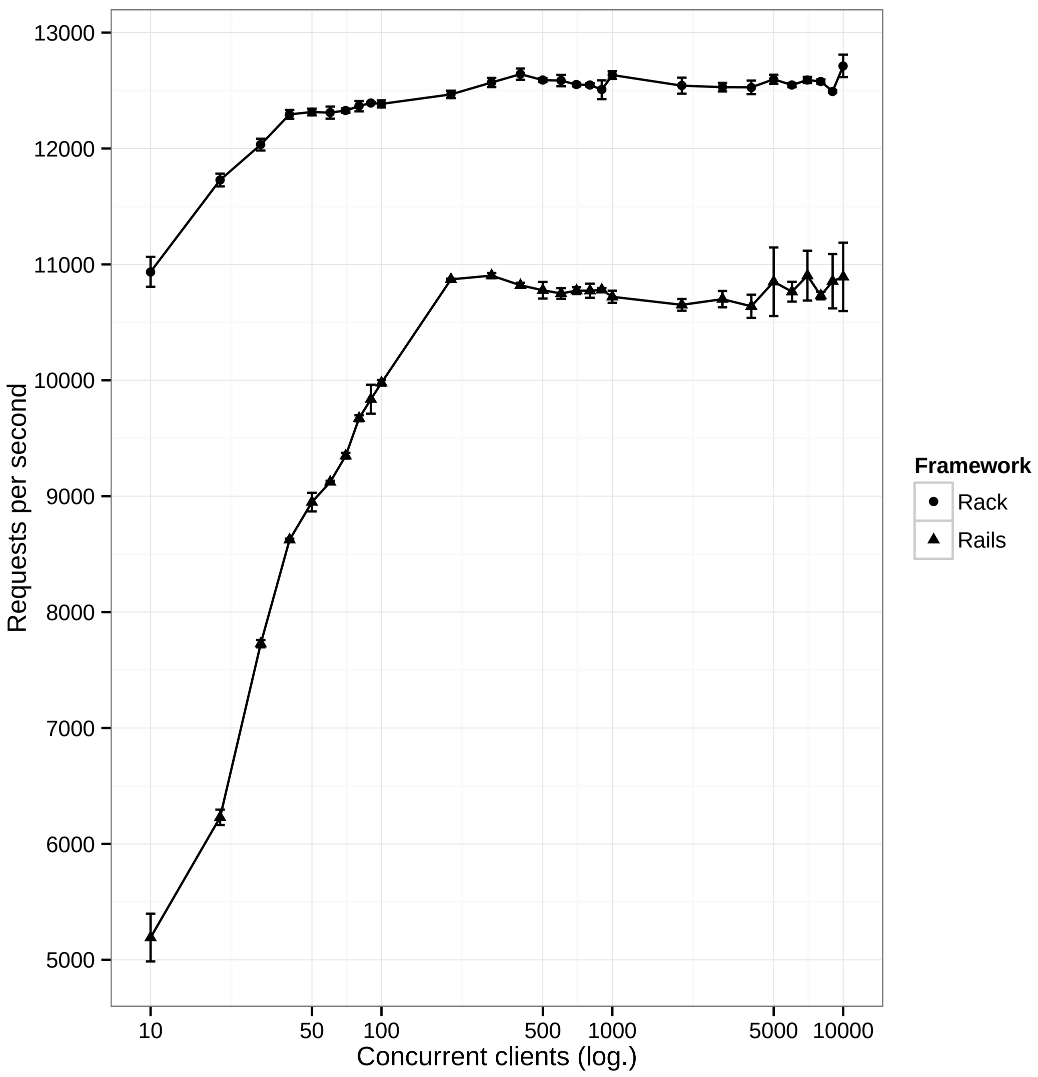

Figure 7.4.: Throughput by Framework in MRI 2.3.0dev

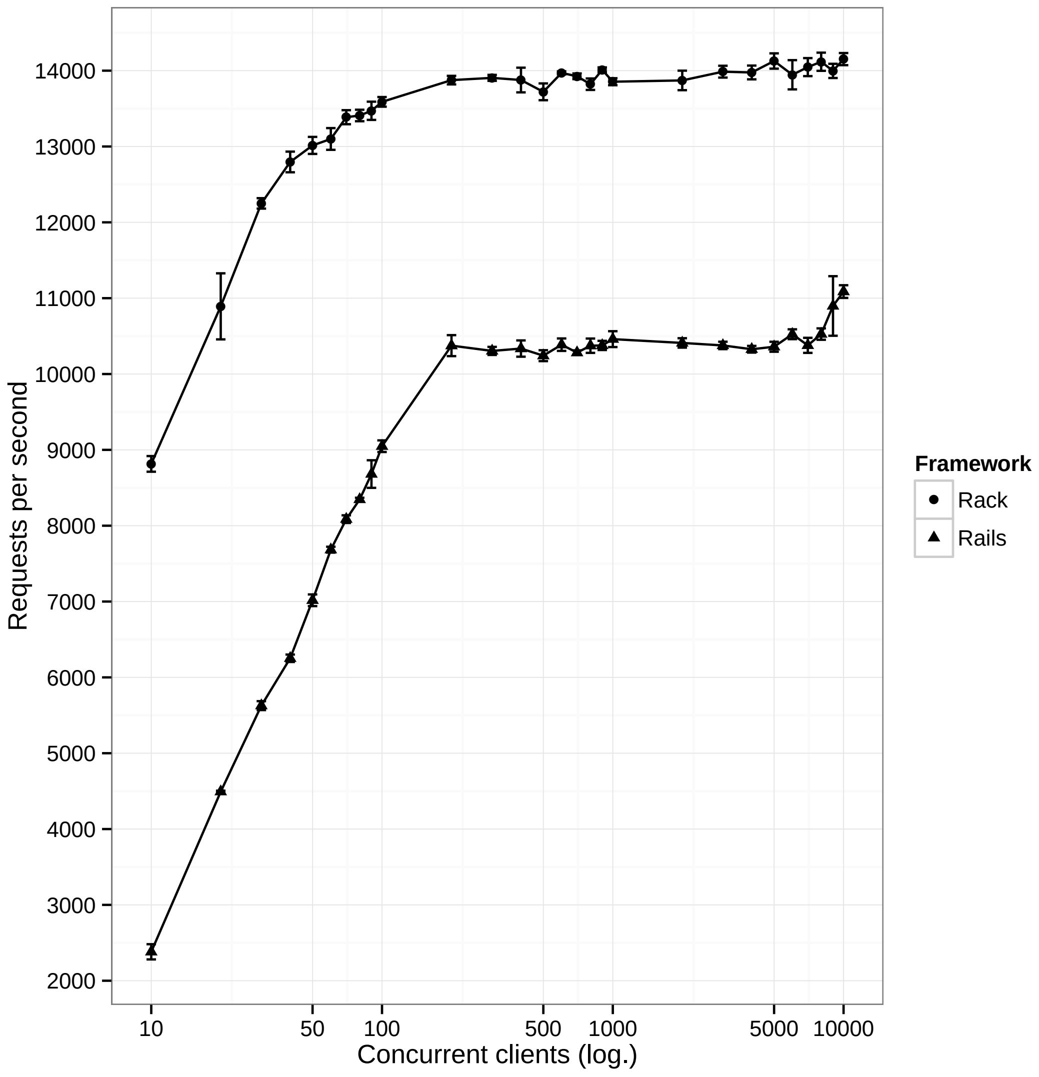

Figure 7.5.: Throughput by Framework in JRuby 1.7.19

The rate of requests per second stays quite stable with rising numbers of concurrent clients in any Ruby VM (see Figure 7.1 and 7.2). In JRuby 1.7.19 throughput rates of about 14,000 requests per second are possible. MRI 2.3.0dev handles about 12,500 requests per second. In MRI with Rack, the peak throughput is reached with quite a low number of concurrent clients compared to Rails. Until 200 concurrent clients, the Rails framework overhead in the event loop seems to be the bottleneck. A decrease in throughput with high numbers of concurrent clients does not occur. In JRuby the throughput with Rack and Rails in relation to concurrent clients runs quite comparable with between 6,500 and 3,500 requests per second deviation. With Rails JRuby 1.7.19 does not perform as good as MRI 2.3.0dev especially with lower numbers of concurrent clients (see Figure 7.4 and 7.5). In JRuby 9.0.0.0-pre1 the server has a surprisingly low throughput especially with Rails and mainly with low numbers of concurrent clients.

Figure 7.3 shows the message loss through timeouts waiting for responses by Ruby VM with Rack. With 300 concurrent clients stressing the server, timeouts when waiting for responses start to occur. This happens throughout different VMs and with different numbers of total messages received during a 30 seconds stressing period. The timeout rates rise with the number of concurrent clients to a maximum of about 9% at 100,000 concurrent clients (MRI 2.2.0p0 serving a Rails application). We tried increasing the UDP receive buffer up to 1 MiB (default was 208 KiB) via the sysctl flag `net.core.rmem_max` but could not observe a descrease in message loss. It raises the question if *Cf-CoAPBench* includes the exponential back-off mechanism for retransmissions of confirmable messages to achieve congestion control (see section 4.2 of [RFC 7252]). We could not find any indication for this (see line 113 of VirtualClient).

Plots showing the message timeouts by Ruby VM with Rails and the throughput by framework in MRI 2.2.0p0 and JRuby 9.0.0.0-pre1 are included in the appendix (see section B.1).

7.2.2. Interoperability

The interoperability of the server has to be evaluated especially regarding to the two interfaces with other software. It interfaces via CoAP with other nodes and via Rack with web frameworks. So test cases are needed for both interfaces that let us specify the interoperability capabilities of the implemented software. The European Telecommunications Standards Institute (ETSI) CoAP Plugtests [17] are specifications for interoperability tests previously conducted within the community of CoAP developers. They are both valuable for testing the CoAP and Rack interoperability. It is possible to analyze the following questions utilizing implementations of this test specifications. However, some of the test cases are not compatible to the specifications of the latest CoAP version [RFC 7252]. `TD_COAP_CORE_11` for example demands a missing token option on a response. Others are not specified detailed enough. `TD_COAP_CORE_13` for example does not require the server to actually handle the URI-Query options.

1. Which CoAP features are supported and possible to implement in an application hosted by the server?
2. How compatible is the Rack interface implementation of the server to different web frameworks?
3. How compatible is the CoAP implementation of the server to different CoAP clients?

Data for answering these questions is gained by means of implementing the Plugtests as RSpec tests. Most Plugtests have been implemented, but not all. We had to concentrate on the tests for the most important aspects regarding to the research questions. Currently, only the CoAP protocol tests 1-13 (without 9) are implemented (see section 7.1 of [17]). `TD_COAP_CORE_09` is not implemented, because the server does not support separate responses. From the optional test case set we implemented the block-wise transfers tests `TD_COAP_BLOCK_01` and `TD_COAP_BLOCK_02` as well as the first three observe tests. Block-wise transfers with `PUT` and `POST` are not supported by the server and therefore also not tested (ETSI Plugtests `TD_COAP_BLOCK_03` and `TD_COAP_BLOCK_04`). The same applies to server observe deregistration detection (tests `TD_COAP_OBS_04` and `TD_COAP_OBS_05`).

The automatic provisioning of the Resource Discovery only works with Rails. Therefore the Plugtests regarding to the CoRE Link Format were omited for this test series. Tests similar to the CoRE Link Format test cases have been implemented before the evaluation phase as unit and integration tests (see for example `spec/resource_discovery_spec.rb` in the david repository). We tested different Rails applications enabling the automatic provisioning of Resource Discovery with the *CoAP crawler client* on `http://coap.me` and there was no indication of problems in the RD middleware implementation. Tested applications were the dummy used for automatic testing of David (see spec/dummy), the core-rd application, and a prototype Rails application used for the development of David. The interoperability of the RD application is not tested as we could not find any RD client implementation or test cases.

Features

During the implementation of the Plugtests, test cases occured that are impossible to be realized because of lacking server support for certain features. All of those shortcomings were already known. Separate responses (see section 5.2.2 of [RFC 7252] and test TD_COAP_CORE_09 of [17]) are one example. Another one is the lack of support for block-wise transfers for incoming messages (see ETSI Plugtests `TD_COAP_BLOCK_03` and `TD_COAP_BLOCK_04`). The token generation of the client library has been revised during the implementation of the test `TD_COAP_CORE_11`. We also found regressions in the server implementation regarding to the support for block-wise transfers and observe. As the Rails application is used solely as an API, we removed all stylesheet or JavaScript asset related gems and paths from the source tree.

Rack

We implemented the Client side of the Plugtests as RSpec tests and the server side with means of different Rack based web frameworks. A list of the tested frameworks and their versions is shown in Table 7.1. Table 3.1 contains an exhaustive list of frameworks and their download locations. The framework specific applications are to be found in the `lib/david/etsi` folder of the david repository.

Framework	Version
Grape	0.10.1
Hobbit	0.6.0
NYNY	2.2.1
Pure Rack	
Sinatra	1.4.5
Ruby on Rails	4.2.0

Table 7.1.: Rack based web frameworks tested on interoperability

When designing the HTTP status code mappings and explicitly returning a CoAP response code by using a Float, `#to_i` was not called on the status. At the time of Rack interface interoperability testing, it became obvious that returning Float status codes does not work anymore from any framework. So two changes were made to the server code. First it is possible that CoAP response codes are used in the style of HTTP response codes (for example *205* meaning CoAP *2.05*. However, with the designed mapping in place, *204* for example would be interpreted as HTTP *204 No Content* and mapped to

CoAP *2.05 Content*. Therefore, a possibility to configure the status code mapping as minimal as possible was implemented. Currently only HTTP *200 OK* maps to CoAP *2.05 Content* in this mode. The Rack environment option `MinimalMapping` can be set to true to activate the minimal mapping. The second change affects the method in Rack code that calls `#to_i` on the status returned by the framework. This method was monkey-patched to omit calling `#to_i` on the status if it is an instance of the Float class. *Grape* and *Sinatra* call `#to_i` on the status by themselves. *Grape* uses `Rack::Request`, which calls `#to_i` on initialization and in the *Sinatra* code it is called in different places. With these frameworks CoAP response codes as Integers have to be used for now. Wrapping the status in a CoAP status object instance is not a trivial solution. The response code *205* for example in HTTP means *Reset Content* and results in clearing the body and headers in *Grape*. The wrapper object would have to actually return an Integer on call of `#to_i`. Code like `status = status.to_i` (like in `Rack::Response`) would erase the CoAP response code information. Out of the tested frameworks, only *Hobbit*, Pure Rack and Rails support the CoAP response codes as Floats.

CoAP

We conduct the Plugtests with different client implementations in connection with the David server hosting the Rack Plugtests applications (see `mandatory/rack.rb` and `optional/rack.rb` in the `lib/david/etsi` directory). Table 7.2 shows the different clients tested, their respective versions and websites. *Californium* and *jcoap* already provide a Plugtests client. *libcoap* ships with a flexible Command Line Interface (CLI) utility that can be used from within scripts. The Ruby coap gem is not listed here, because it is already tested on protocol interoperability in connection with the Rack interface interoperability tests (see section 7.2.2).

Client	Version	Source
Californium	1.0.0-M3	`https://github.com/eclipse/californium`
Copper	0.18.4	`https://addons.mozilla.org/de/firefox/addon/copper-270430`
libcoap	4.1.1	`http://sourceforge.net/projects/libcoap`

Table 7.2.: Clients used for CoAP interoperability tests

For an extensive summary of the test results, see Table 7.3. All chosen Plugtests pass with the *cf-plugtest-checker* component of *Californium* except two observe tests. A manual test series with *Copper* yielded in positive results for all tests. *libcoap* was also tested manually with the `coap-client` utility (see examples/client.c). In some test cases options had to be passed on the command line manually with the `-O` switch. `TD_COAP_OBS_02` (which tests observe deregistration) is an example. The interoperability of the transparent Resource Discovery was manually tested with *Copper* without strictly following the Plugtests.

	CORE												BLOCK		OBS		
	01	02	03	04	05	06	07	08	10	11	12	13	01	02	01	02	03
Californium	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×
Copper	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
libcoap	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 7.3.: CoAP interoperability test results

7.2.3. Reflection

Performance

The throughput of JRuby is not much higher than of MRI. This can be explained by the handling of messages in a single threaded event loop. To use the full potential of JRuby, probably a thread pool for message handling could be beneficial. With *Celluloid*, the supervision of one actor or a pool of actors is quite similar. It should be possible to change the source code of David so that it can better utilize the multithreading capabilities of JRuby. In general, the message loss is higher with Rails than with Rack. This is probably due to the bigger framework overhead inside the event loop. In a Ruby VMs such as JRuby, the performance could possibly be increased by using a thread pool for framework answers or even message parsing.

As we expected, the throughput of David lags behind the throughput of *Californium*. We did not verify the measurements by running the bechmarks for *Californium* on our own hardware, because Matthias Kovatsch conducted performance tests for his dissertation on quite a similar CPU (see section 4.4.2 of [24]). *Californium* handles a throughput of about 40,000 RPS on a single core and up until 135,000 RPS on four cores. Comparing the single core throughput, our Ruby solution reaches about one third of the throughput of *Californium* (2.86 times slower).

Interoperability

There were no tested features not possible to be implemented in the Rack applications. The Rack interface therefore seems to suffice the characteristics of CoAP and its extensions. All tested Rack compatible web frameworks pass the interoperability tests. The usage of Floats to explicitly return CoAP status is – as beforehand mentioned – the only problem to be respected in this context. Although the implemented server lacks some features and the respective interoperability tests with external clients had to be omited, the majority of the actually conducted tests were positive.

7.3. Developer Feedback

The feedback of developers taking an interest in a CoAP server with a Rack interface would provide valuable information for example on usability and bugs of the software created during this work. We tried getting feedback by announcing it publicly on related mailing lists and by conducting an observation of a developer trying to solve a task chosen by himself.

7.3.1. Public Announcement

We wrote a basic development centric README file for David (see `README.md` for the 0.4.0 release[9]) as a central pragmatic documentation. The text covers the most important aspects of usage and information like configuration options. Afterwards we posted an announcement of David with a short description to `rack-devel@googlemail.com` and `rails-talk@googlemail.com` as well as a link to the GitHub repository in the *ruby* subreddit (see subsection B.3.1). There were neither direct answers to the mail posted on the mailing lists nor to the reddit posting. However, GitHub shows a slightly increased traffic to the repository on the days after posting[10]. At the time of the work the only explicit response to this announcement is a bug report[11] and five GitHub users marking the repository as a favorite. The coap repository also has five stargazers. One user reported a bug regarding to the initialization of multicast communications on OS X. The bug could not be reproduced in a VM running Mac OS X 10.9.5 but was fairly straight forward to fix, anyway.

7.3.2. Developer Observation

To gain data regarding to the usability of David for Ruby programmers and to find bugs, an observation of a programmer familiar with Rails and the basics of CoAP developing a self-chosen prototype application has been conducted. We only intervened with specific directions when the process got stuck. The project's README file was the main documentation source. A detailed log of the experiment resides in subsection B.3.2. We will concentrate on the results here. As we only observed a single developer once and without a reproducable task, we do not claim a statistically significant statement by the experiment but an approach to get feedback.

The test subject started with reading the David README and decided on implementing a basic Rails application returning JSON data possibly transcoded into CBOR. During the process of realization, some issues with the documentation and usage arose and some bugs in the server software occured. The documentation at the point of the experiment lacked especially introductory passages for users without advanced knowledge of Rack and CoAP. The most important issues with the code arosen during the observation are as follows.

1. At the time of the experiment, David does not cache a framework answer even

[9] `https://github.com/nning/david/blob/0.4.0/README.md`
[10] `https://github.com/nning/david/graphs/traffic`
[11] `https://github.com/nning/david/issues/1`

when it is requested through block-wise transfers. If the payload changes between different requests with Block2 Option in control usage (see section 2.3 of [core-block-17]), the ETag value changes and the client has to try again to obtain a fully consistent resource representation. Framework answers could be cached at least per endpoint so that this retrial becomes unnecessary.

2. HTTP Chunked Transfer Coding is not implemented to be reassembled, because during the server development it was only used in certain corner cases (for example using `rackup` with Rails in development mode).

3. Outgoing JSON/CBOR transcoding did not work at the time of the experiment, because it broke earlier in a way not covered by the unit tests. This was fixed right away.

4. With activated transcoding and the Accept option of a request set to 'application/cbor' (60), a CoAP *4.06 Not Acceptable* is responded by the framework, because it the transcoding is transparent. I case of an activated transcoding also headers like the Accept header have to be rewritten.

The test subject explicitly stated that the server works well as a drop-in replacement even without advanced knowledge of CoAP specifics.

Chapter 8

Conclusion

8.1. Recapitulation of Results

The following subsection lists our accomplishments and the deficiencies of our implementation for each research question.

How can CoAP be implemented?

We iteratively designed and developed the protocol handling for a CoAP server. Opposed to our previous estimations, we did not depend on larger parts of a shared library for the CoAP transmission layer but only for message parsing and assembly. The implementation makes use of Message Deduplication to lower the waste of server resources through retransmitted confirmable messages. Resource representations are requestable utilizing Block-wise transfers. Our client interoperability evaluation showed a very good protocol interoperability, although we would favor a more polished CoAP transmission implementation that supports separate responses, confirmable messages initiated by the server, and incoming Block-wise Transfers. A structure comparable to Rack's middleware stack for different protocol layers would probably have improved code modularity and quality.

How to handle a huge amount of IoT devices?

Message handling was implemented utilizing a single threaded, non blocking event loop as a model providing a good balance between performance and Ruby VM interoperability. In the performance evaluations, our realization of this model yielded a message throughput almost three times higher than previously anticipated. Also, our server has almost twice the throughput as Ruby HTTP servers running on MRI. However, further performance optimizations that make closer use of the different Ruby VM characteristics would be desirable.

Does the Rack interface comply with the characteristics of CoAP?

We showed with our implementation of the Rack interface that it is basically also working in a sufficient way in conjunction with CoAP. Some functionality was implemented as Rack middleware by us (`David::ShowExceptions`, `David::ResourceDiscovery`). The framework interoperability tests showed that the server implementation of the Rack interface is compatible to numerous different Rack based web frameworks. The only major restriction of the Rack interface is the problematic integration of asynchronous responses. The performance of our implementation in average cases could probably be improved by integrating a caching of Rack application responses.

How can a later integration of transport encryption with DTLS [RFC 6347] be supported?

We provide a basic interface for a DTLS integration and means to pass details of the encrypted connection to the Rack middleware stack and the application.

How can headers be translated between HTTP and CoAP?

According to our design, the server transparently translates CoAP requests into Rack environments and Rack application responses into CoAP responses. The framework and client interoperability tests showed that this translation works well. Only the explicit returning of CoAP response codes, while a status code mapping from HTTP to CoAP is activated, keeps to be somewhat cumbersome. We also did not implement all possible translations as Conditional Request Options for example (see section 5.10.8 of [RFC 7252]).

How can payload formats be translated between their suitable fields of application?

Our design and implementation provides an automatic and transparent transcoding between JSON and CBOR resource representations. However, it is only activatable for an controller or action of the application. We also would have liked a more extensive and generalized transcoding implementation as a Rack middleware.

Is the CoAP Resource Discovery implementable transparently for the Rack application?

We provide a solution for a transparent discovery of the resources of a Rails application as a Rack middleware. In contrast to most other CoAP implementations, we integrated multicast communications also supporting the CoAP multicast groups. The attributes of resources like interface descriptions are configurable from the Rails controllers. Unfortunately, we could not provide a Rack application and framework agnostic way of resource autodetection.

Can a Resource Directory be realized with the created software?

The RD and Lookup Funtion Sets of the RD draft [core-resource-directory-02] have been implemented as a Rails application by us. The development process showed that our solution is compatible to the specifications of that draft and that TDD with RSpec works for CoAP application development with Rails. However, the Group Function Set is not supported yet. The Lookup Function Set was only implemented for resource lookup.

Is it possible to support Observe?

We implemented the Observe extension based on *Celluloid* actors. The determination whether updates are necessary is solved utilizing entity tags. We also designed a system based on HTTP Caching to minimize the overhead of update checks. Our interoperability evaluations showed that the implemented observe support works with different clients and frameworks. Some implementation details such as garbage collection of observe relationships, caching on update checks, and deregister through reset messages were not realized by us.

8.2. Perspectives

Implementing solutions to the deficiencies mentioned in the preceding accomplishment summary is a next step for the further existence of the created software system as an Open Source project. Especially the polishing of the CoAP transmission implementation, further performance optimizations, working DTLS, and protocol translations in a more detailed way are important goals. For performance optimizations the examination of the architectures AMPED, SEDA and PIPELINED would be interesting (see section 4.1.2 of [24]). The transcoding functionality of the server should be decoupled, generalized to work with other content formats, and – as a Rack middleware – made independent of CoAP. The possibility to activate transcodings for single controllers or actions would be a useful feature. As the autodetection of resources for the Resource Discovery currently only works with Rails, compatiblity to other Rack compatible frameworks would be an improvement. As mentioned before, the RD application and the observe implementation are also providing some possibilities to be further improved. We will further develop and maintain the open source projects started during this work and probably try to get other people involved. That includes the Ruby CoAP library for which also numerous improvements are possible.

Beyond the concrete implementation created during this work, there are some possible applications of parts of our design in the context of other CoAP servers. We would like to see solutions for hosting IoT web applications developed in Ruby comparable to nginx[1] and Phusion Passenger[2] for example as well as CoAP servers compatible to web frameworks written in programming languages optimized for concurrency such as elixir[3] for example.

[1] http://nginx.org
[2] https://www.phusionpassenger.com
[3] http://elixir-lang.org

Appendix A

Guides

A.1. Basic Installation and Usage

It is assumed that a recent version of Ruby is installed. At the time of writing, Ruby 2.2.0 is the latest stable version released. Documentation about how to install Ruby on different OSs can be obtained from the official Ruby website[1].

In case a new Rails application is to be used, it is created with the following command.

```
rails new example
```

If a Rails application is existent, David can be included as a dependency by including the following line into `Gemfile`.

```
gem 'david'
```

David will automatically hook into Rails to make itself the default Rack handler so that `rails s` starts David by default. If the application has to be used via HTTP, WEBrick can be started by executing `rails s webrick`. After the server is started, the Rails application is available at `coap://[::1]:3000/` by default.

Copper is a CoAP client for Firefox and can be used for development. The Ruby coap gem is used by David for example for message parsing and also includes a command line utility (named `coap`) that can also be used for development.

As CoAP is a protocol for constrained environments and machine to machine communications, returning HTML from controllers will not be of much use. JSON for example is more suitable in that context. David works well with the default ways to handle JSON responses from controllers such as `render json:`. You can also utilize Jbuilder templates for easy generation of more complex JSON structures.

CBOR [RFC 7049] can be used to compress JSON documents. Automatic transcoding between JSON and CBOR is activated by setting the Rack environment option `CBOR` or `config.coap.cbor` in your Rails application config to `true`.

[1]https://www.ruby-lang.org/en/documentation/installation/

A.2. Running the Resource Directory (RD)

Clone the repository.

```
git clone https://github.com/nning/core-rd.git
cd core-rd
```

Install dependencies and run database migrations.

```
gem install bundler
bundle --deployment --without 'development doc test'

export RAILS_ENV=production
rake db:migrate
rake db:seed
```

Generate a secret token for the production environment. It has to be pasted in the production section of `config/secrets.yml`.

```
rake secret
```

Start the server.

```
rackup
```

Appendix B

Raw Evaluation Data

B.1. Additional Performance Benchmark Plots

Figure B.1, B.2, and B.3 were kept out of subsection 7.2.1, because they do not add significant information to the analysis. They are included in the appendix for the sake of completeness.

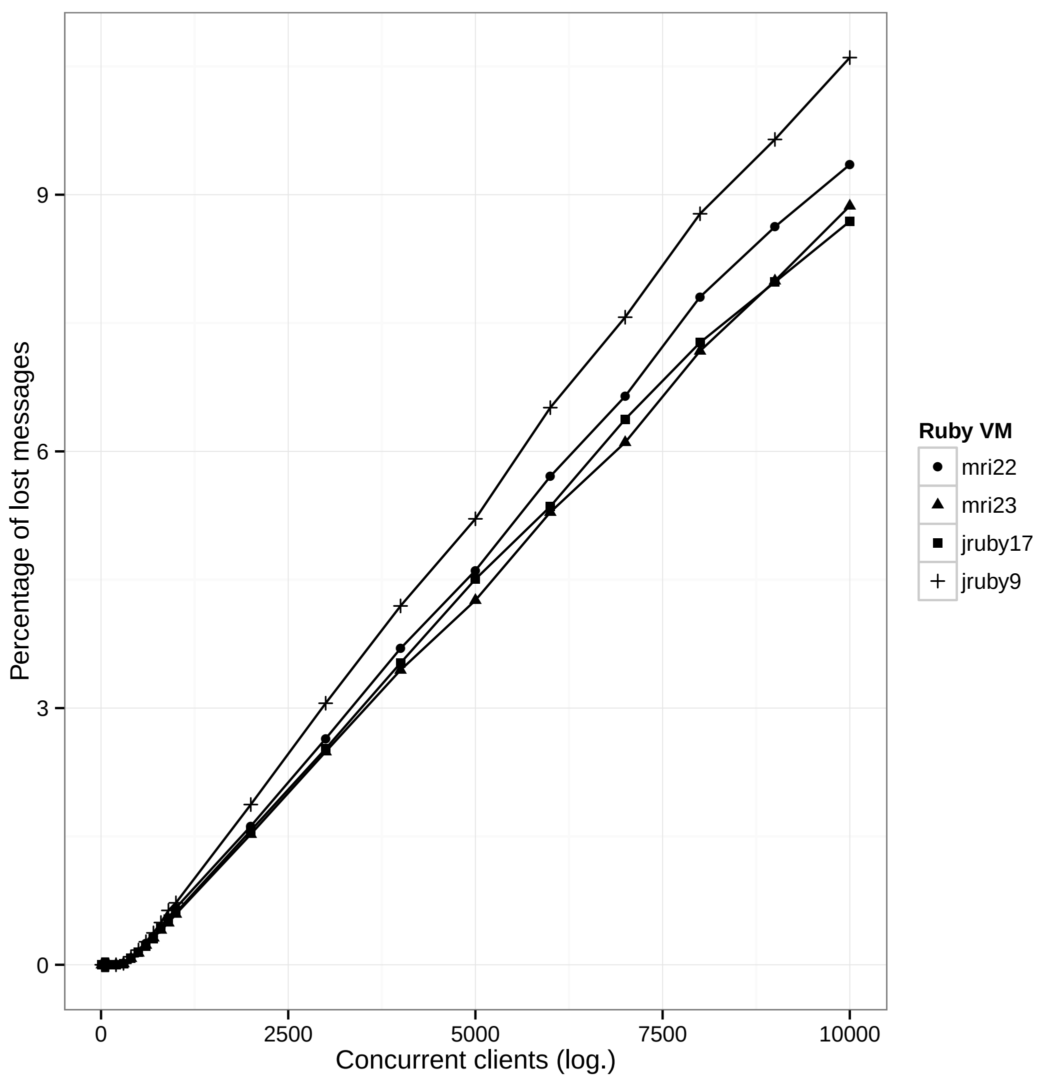

Figure B.1.: Message timeouts by Ruby VM with Rails

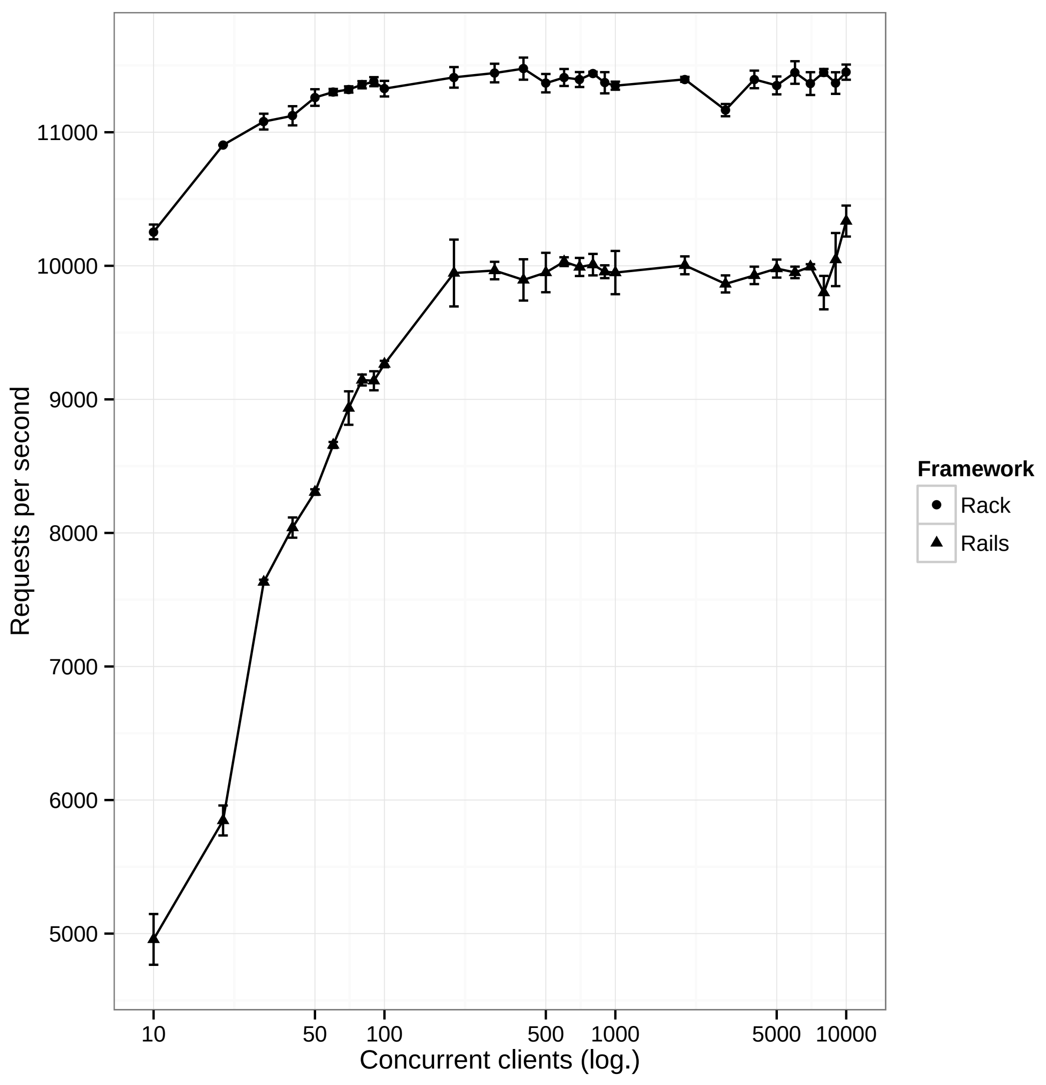

Figure B.2.: Throughput by Framework in MRI 2.2.0p0

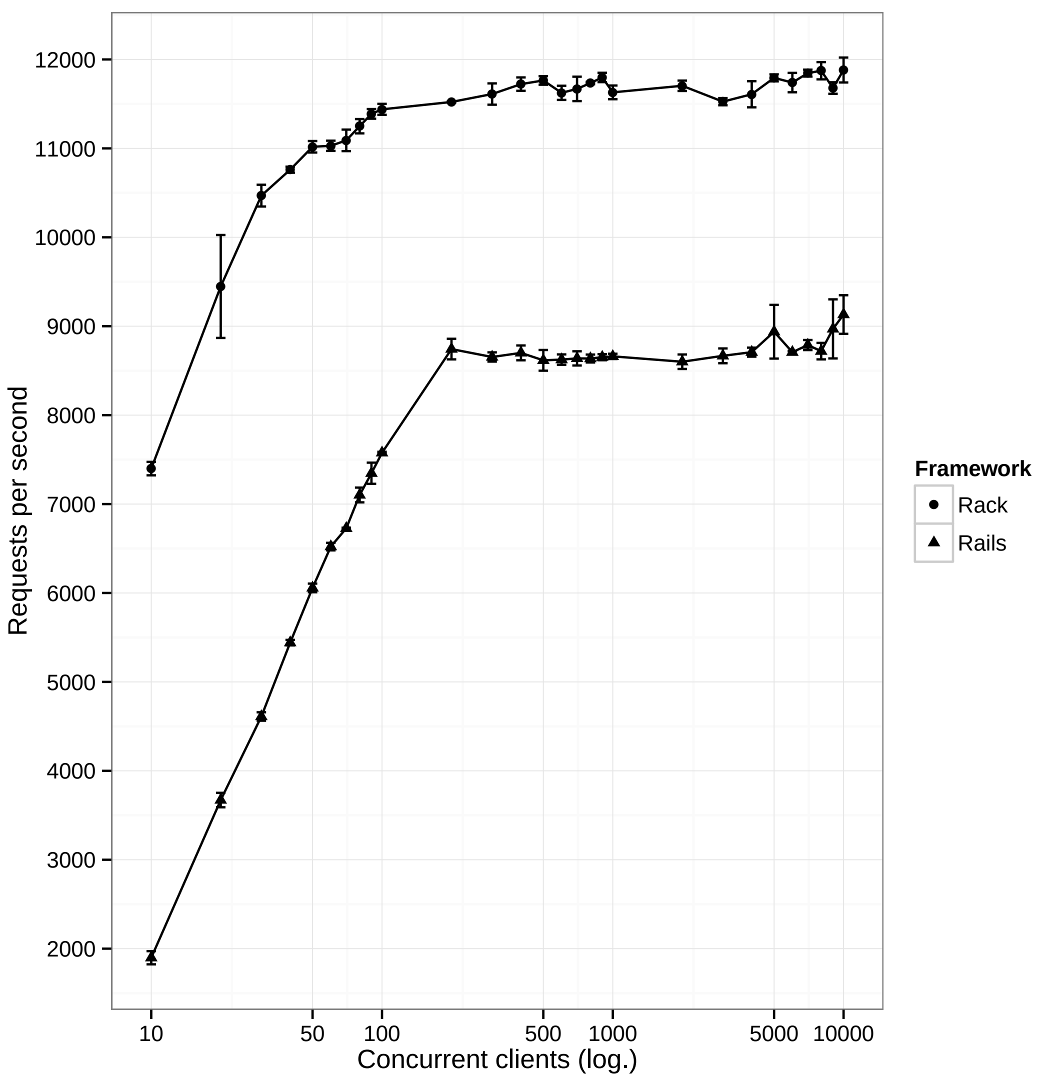

Figure B.3.: Throughput by Framework in JRuby 9.0.0.0-pre1

B.2. Performance Benchmark on Ubiquiti EdgeMAX Lite

B.2.1. Router Preperations

Firmware Update

- Reset

- Firmware update to EdgeOS 1.6.0 (2014-11-05, Debian wheezy)

Web Interface

After the reset the router has the IP address `192.168.1.1` on the first ethernet interface. Both login and password for the webinterface is `ubnt`. The performed configuration steps are:

- DHCP client on eth0 and eth2
- Deactivation of DHCP server and NAT

B.2.2. Installing Ruby and git

SSH and Root Shell

```
ssh ubnt -l ubnt
sudo -s
```

Software Repositories

Both Debian stable and testing repository sources are needed.

Listing B.1: /etc/apt/sources.list.d/stable.list

```
deb http://ftp.de.debian.org/debian/ stable main contrib non-free
deb http://security.debian.org/ stable/updates main contrib non-↩
    free
```

Listing B.2: /etc/apt/sources.list.d/testing.list

```
deb http://ftp.de.debian.org/debian/ testing main contrib non-↩
    free
deb http://security.debian.org/ testing/updates main contrib non-↩
    free
```

After configuring the package sources, `apt-get update` has to be executed.

Install Packages

The installation of git and essential tools to compile gems written in C is accomplished with the following command. We install gcc 4.9.1 from testing, because version 1.8.2 of the the json gem needs the `fstack-protector-strong` option and it is not supported in gcc 4.6.3. Also we could not get the cbor gem compiling in gcc 4.6.3 on mips.

```
apt-get install -y git-core build-essential libsqlite3-dev
apt-get install -y -t testing gcc
```

Ruby has to be installed from testing (Ruby 2.1.5p273 and rubygems 2.2.2). Although it would be possible to use Ruby 1.9.3p194 from stable, the VM segfaulted (only on MIPS) in connection to *Celluloid*. We could not get newer versions of Ruby building with rbenv on this platform.

```
apt-get -y -t testing install ruby ruby2.1 ruby2.1-dev
```

PATH for Gem Binaries

```
echo "export PATH#\"\$PATH:/usr/local/bin\"" >> ˜/.bashrc
source ˜/.bashrc
```

B.2.3. Using David for Performance Benchmarks

To clone the code, issue the following commands. This guide was tested at commit 529c73c.

```
cd /srv
git clone https://github.com/nning/david.git
cd david
```

Generating documentation for each installed gem can be turned of with some configuration in `/.gemrc`.

```
cat > ˜/.gemrc <<EOF
install: --no-rdoc --no-ri
update: --no-rdoc --no-ri
EOF
```

The dependencies minimally necessary for production use are installed with bundler.

```
gem install bundler
cd benchmarks/rackup
bundle --without ’grape rails’
```

Celluloid will throw a `Celluloid::FiberStackError` *stack level too deep* exception on server startup if the Ruby VM stack size for Fiber creations is not sufficient. It is configurable with the environment variable `RUBY_FIBER_VM_STACK_SIZE`. In our test, it was set to 64 KiB by default on this hardware. To increase it to 128 KiB (for the current shell session) run the following command.

```
export RUBY_FIBER_VM_STACK_SIZE=131072
```

The server is started with the performance benchmark Rack application as follows.

```
rackup rack.ru
```

At this stage, the Ruby VM unfortunately segfaulted in connection with the timers gem used by *Celluloid*.

JRuby as an Alternative

We tried to get the performance benchmark setup running in JRuby. JRuby 1.5.6 is available in the package repositories of Debian wheezy. That version is from 2010 and was previously untested with David. In theory the following shell commands should install JRuby and its dependencies, and start the Rack performance benchmark application after installing its dependencies. As with MRI, we could not get the application running. The command installing the bundler gem (see line 2 of the shell commands below) utilizes the CPU at 100% and seemed to be hanging without ever finishing only to result in a `java.lang.ArrayIndexOutOfBoundsException`.

```
1 apt-get install -y default-jre-headless jruby
2 jruby -S gem install bundler
3 cd /srv/david/benchmarks/rackup
4 jruby -S bundle --without 'grape rails'
5 jruby -S rackup rack.ru
```

B.3. Developer Feedback

B.3.1. Public Announcement

On the 5th of February 2015, we announced David to the Rack development mailing list `rack-devel@googlemail.com`[1] and posted a link to the GitHub repository to the ruby subreddit[2]. Listing B.3 depicts the original mail posted to that list. On the next day, we also posted this text to the public Rails discussion mailing list `rails-talk@googlemail.com`[3].

[1] https://groups.google.com/forum/#!topic/rack-devel/vKy9eaoiLxI
[2] https://www.reddit.com/r/ruby/comments/2uvk6n/a_coap_server_with_a_rack_interface_for_use_of/
[3] https://groups.google.com/forum/#!topic/rubyonrails-talk/5m8ex-oGQvs

Listing B.3: Announcement of David on mailing lists

```
From: spam@orgizm.net
To: rack-devel@googlemail.com
Date: Tue, 05 Feb 2015 16:15:55 +0100
Subject: CoAP server with Rack interface

Hey,

I’m working on a CoAP [0] server with a Rack interface for my
diploma thesis:

https://github.com/nning/david

Your feedback would be very valuable for me although it is not
clear, I manage to make many changes based on it before handing
in the thesis. I compiled a quick README and hope it suffices
as a starting point for testing the server. Maybe it is just
interesting, the Rack interface is used in another protocol
context.

Some more info on CoAP from RFC7252:

The Constrained Application Protocol (CoAP) is a specialized
web transfer protocol for use with constrained nodes and
constrained (e.g., low-power, lossy) networks. The protocol is
designed for machine- to-machine (M2M) applications such as
smart energy and building automation.

CoAP provides a request/response interaction model between
application endpoints, supports built-in discovery of services
and resources, and includes key concepts of the Web such as
URIs and Internet media types. CoAP is designed to easily
interface with HTTP for integration with the Web while meeting
specialized requirements such as multicast support, very low
overhead, and simplicity for constrained environments.

Thanks!
henning

[0] https://tools.ietf.org/html/rfc7252
```

B.3.2. Developer Observation

We asked Ingo Becker, a programmer with a background in web application and microcontroller development to try realizing a prototype CoAP application with Rails and David. He is familiar with the former and knows the basics of CoAP. He was working on his own notebook and we tried to only intervene minimally, taking notes about the procedure and problems. When the process got stuck, we helped with specific directions. As quite usual in the Rails world, he mostly relied on the README file from the GitHub repository[4] for documentation. The following list contains the raw notes from the observation. Sentences in square brackets are annotations and associations of the observer. Serious issues are in bold font.

- Which Ruby versions are supported? [Ruby 2.1.1p76 pre-installed.]
- [Rails 4.1.6 pre-installed. `rails new foo` created Rails 4.1.6 application. Adding `gem 'david'` and executing `bundle update` (without `--without test`) also installs Rails 4.2.0.]
- [Examined David version is 0.4.0.]
- [Repeated use of "You can" in first paragraphs of README.]
- Include hint to `render json:` and the responders gem before the notes on Jbuilder.
- [Mention coap CLI utility from coap gem in addition to Copper.]
- Introduction to Tested Rack Frameworks.
- Introduction how to use Rack options with `rails s` (or maybe also `rackup`.
- What are Block-wise Transfers?
- Maybe documentation on removing JavaScript asset related gems.
- [Examine `CoRE::Link` regarding to obs attribute.]
- Output of CoAP URI for copy and pasting after David startup.
- `GET /` results in missing template (JSON formatted *welcome#index*).
- Documentation on `gem install cbor` or revise message if it is not installed.

[4]https://github.com/nning/david

- Documentation on JSON as default value for HTTP Accept header.
- README should include hints for starting server in a way, the actual messages can be examined [with debug log level].
- Document context for mentioned configuration options.
- [Maybe include a generator that patches for example `config.ru` to start on port 5683/udp.]
- **If David calls the Rack application for every block of a response representation, what happens if payload changes in between a block-wise transaction?** [The ETag option is changed and the client can try to obtain the full representation, again.]
- There is *Celluloid* log output after execution of `rails g controller` and `rails c` and after the former the process even hangs for ten seconds before exiting.
- [Reads Jbuilder documentation.]
- [Adds route to index action of exemplary controller and renders minimal static JSON document with a Jbuilder template.]
- [Starts `rails s webrick` to test routing, because log output with block-wise transfers of an exception message is confusing. Examines HTTP headers. Starts CoAP server.]
- [Talking about debug log level. Starts server with `rackup` to specify log level.] Known bug: **With `rackup` in development mode, Rails answers chunked but David does not reassemble the chunks before responding via CoAP.**
- [Subject wants to try JSON/CBOR transcoding, because it was mentioned in README file.]
- Bug: **Outgoing transcoding does not work; JSON is responded.** [Fixed directly and switched from David 0.4.0 to master.]
- README should state that choosing a bigger block size in client creates less noise in server log when debugging.
- Bug: **4.06 is returned when explicitly setting `Accept: application/cbor` in Copper (despite transcoding is activated).**

- [Talking about 6lobac [6lo-6lobac-00].]
- States, the server is working well as a drop-in replacement even without advanced knowledge of CoAP.

List of Figures

5.1. Exemplary actor interaction on *Observe tick* 30
5.2. Interaction regarding action annotation for Resource Discovery 40
5.3. Observe interaction with conditional requests 43

6.1. Object Relations (Server) 55
6.2. Object Relations (Middleware) 56

7.1. Throughput by Ruby VM with Rack 62
7.2. Throughput by Ruby VM with Rails 63
7.3. Message timeouts by Ruby VM with Rack 64
7.4. Throughput by Framework in MRI 2.3.0dev 65
7.5. Throughput by Framework in JRuby 1.7.19 66

B.1. Message timeouts by Ruby VM with Rails 84
B.2. Throughput by Framework in MRI 2.2.0p0 85
B.3. Throughput by Framework in JRuby 9.0.0.0-pre1 86

List of Tables

3.1. Rack frameworks 12
3.2. CoAP Libraries – Possible compatiblity with Ruby and multicast support 14
3.3. CoAP Libraries – Examined versions 15

5.1. Mapping of HTTP to CoAP return codes (*2xx Success*) 34
5.2. Mapping of HTTP to CoAP return codes (*4xx Client Error*) 35
5.3. Direct mappings from CoAP options to HTTP headers 36

6.1. Implemented Components 45
6.2. External Libraries 45
6.3. David configuration options 49
6.4. Rack environment additions of David 49

7.1. Rack based web frameworks tested on interoperability 69
7.2. Clients used for CoAP interoperability tests 70
7.3. CoAP interoperability test results 72

Acronyms

AMPED	Asynchronous Multi-Process Event-Driven
API	Application Programming Interface
BDD	Behavior-Driven Development
BSON	Binary JSON
CLI	Command Line Interface
CBOR	Concise Binary Object Representation
CGI	Common Gateway Interface
CI	Continuous Integration
CoAP	Constrained Application Protocol
CoC	Convention over Configuration
CoRE	Constrained RESTful Environments
CPU	Central Processing Unit
DICE	DTLS In Constrained Environments
DRY	Don't Repeat Yourself
DSL	Domain Specific Language
DTLS	Datagram Transport Layer Security
ETSI	European Telecommunications Standards Institute
EXI	Efficient XML Interchange
FSM	Finite-state machine

GIL Global Interpreter Lock

GPL General Public License

GWT Google Web Toolkit

HTML HyperText Markup Language

HTML5 HyperText Markup Language, Version 5

HTTP HyperText Transport Protocol

IETF Internet Engineering Task Force

IoT Internet of Things

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

JSON JavaScript Object Notation

JVM Java Virtual Machine

M2M Machine-to-Machine

MRI Matz's Ruby Interpreter

MVC Model-View-Controller

OS Operating System

Rails Ruby on Rails

RAM Random-access Memory

RD Resource Directory

REST Representational State Transfer

ROM Read-only Memory

RPS requests per second

SEDA Staged Event-Driven Architecure

SPED Singe-Process Event-Driven

TCP Transmission Control Protocol

TDD Test-driven Development

TLS Transport Layer Security

UBJSON Universal Binary JSON

UDP User Datagram Protocol

URI Uniform Resource Identifier

VM Virtual Machine

WG Working Group

XML eXtensible Markup Language

Bibliography

[1] C. Bormann et al. *CBOR (RFC 7049) extension for Ruby*. URL: https://github.com/cabo/cbor-ruby (visited on 03/09/2015) (cit. on pp. 37, 54).

[2] *Are all Ruby built-in objects thread safe?* URL: https://www.ruby-forum.com/topic/174086 (visited on 03/09/2015) (cit. on p. 31).

[3] B. Batsov. *The Ruby Style Guide*. URL: https://github.com/bbatsov/ruby-style-guide (visited on 03/09/2015) (cit. on p. 26).

[httpbis-http2-17] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol version 2*. draft-ietf-httpbis-http2-17. IETF, Feb. 2015. URL: https://tools.ietf.org/html/draft-ietf-httpbis-http2-17 (cit. on p. 16).

[4] O. Bergmann. *libcoap: C-Implementation of CoAP*. URL: http://sourceforge.net/projects/libcoap (visited on 03/09/2015) (cit. on p. 5).

[5] C. Bormann, A. P. Castellani, and Z. Shelby. "CoAP: An Application Protocol for Billions of Tiny Internet Nodes". In: *Internet Computing, IEEE* 16.2 (Mar. 2012), pp. 62–67. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.29. URL: http://dx.doi.org/10.1109/MIC.2012.29 (cit. on p. 3).

[RFC 7228] C. Bormann, M. Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. RFC 7228. IETF, May 2014. URL: https://tools.ietf.org/html/rfc7228 (cit. on p. 5).

[RFC 7049] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. IETF, Oct. 2013. URL: https://tools.ietf.org/html/rfc7049 (cit. on pp. 9, 17, 21, 36, 81).

[core-block-17] C. Bormann and Z. Shelby. *Block-wise transfers in CoAP*. draft-ietf-core-block-17. IETF, Mar. 2015. URL: https://tools.ietf.org/html/draft-ietf-core-block-17 (cit. on pp. 3, 6, 13, 14, 19, 27, 35, 50, 75).

[6] C. M. Bowman et al. "The Harvest Information Discovery and Access System". In: *Comput. Netw. ISDN Syst.* 28.1-2 (Dec. 1995), pp. 119–125. ISSN: 0169-7552. DOI: 10.1016/0169-7552(95)00098-5. URL: http://dx.doi.org/10.1016/0169-7552(95)00098-5 (cit. on p. 29).

[RFC 7159] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. IETF, Mar. 2014. URL: https://tools.ietf.org/html/rfc7159 (cit. on p. 21).

[7] *Californium (Cf) CoAP framework*. URL: https://eclipse.org/californium (visited on 03/09/2015) (cit. on p. 5).

[8] A. P. Castellani et al. "WebIoT: A web application framework for the internet of things." In: *WCNC Workshops*. IEEE, 2012, pp. 202–207. ISBN: 978-1-4673-0681-2. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6215491 (cit. on p. 5).

[core-http-mapping-06] A. Castellani et al. *Guidelines for HTTP-CoAP Mapping Implementations*. draft-ietf-core-http-mapping-06. IETF, Mar. 2015. URL: https://tools.ietf.org/html/draft-ietf-core-http-mapping-06 (cit. on pp. 7, 33, 36).

[9] *Celluloid*. URL: https://celluloid.io (visited on 03/09/2015) (cit. on pp. 6, 10, 29).

[10] *Celluloid README*. URL: https://github.com/celluloid/celluloid/blob/v0.16.0/README.md (visited on 03/09/2015) (cit. on p. 10).

[11] *Celluloid::IO*. URL: https://github.com/celluloid/celluloid-io (visited on 03/09/2015) (cit. on pp. 10, 27, 29).

[12] *CoAP – Constrained Application Protocol — Implementations*. URL: http://coap.technology/impls.html (visited on 03/09/2015) (cit. on pp. 6, 14).

[13] *ConcurrentHashMap (Java Platform SE 7)*. URL: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html (visited on 03/09/2015) (cit. on p. 15).

[14] *Constrained Application Protocol – Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/CoAP#Implementations (visited on 03/09/2015) (cit. on pp. 6, 7, 14).

[RFC 5246] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. IETF, Aug. 2008. URL: https://tools.ietf.org/html/rfc5246 (cit. on p. 7).

[PEP 333] P. J. Eby. *Python Web Server Gateway Interface v1.0*. Mar. 2011. URL: https://legacy.python.org/dev/peps/pep-0333 (cit. on p. 6).

[15] B. Erb. *Concurrent Programming for Scalable Web Architectures*. Diploma Thesis. Apr. 2012. URL: https://berb.github.io/diploma-thesis (visited on 03/09/2015) (cit. on p. 9).

[16] *Erbium (Er) REST Engine and CoAP Implementation for Contiki*. URL: http://people.inf.ethz.ch/mkovatsc/erbium.php (visited on 03/09/2015) (cit. on p. 5).

[17] *ETSI CTI Plugtests Guide First Draft V0.0.15 (2012-02)*. Feb. 2012. URL: http://www.etsi.org/plugtests/CoAP/Document/CoAP_TestDescriptions_v015.pdf (visited on 03/09/2015) (cit. on pp. 67–69).

[RFC 6455] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. IETF, Dec. 2011. URL: https://tools.ietf.org/html/rfc6455 (cit. on pp. 16, 41).

[REST] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (visited on 03/09/2015) (cit. on p. 2).

[RFC 7233] R. Fielding, Y. Lafon, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. RFC 7233. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7233 (cit. on p. 36).

[RFC 7234] R. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7234 (cit. on p. 42).

[RFC 7232] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7232 (cit. on pp. 11, 21, 34, 42).

[RFC 7230] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7230 (cit. on pp. 2, 16, 38).

[RFC 7231] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7231 (cit. on pp. 2, 10, 11, 21, 33, 34, 36).

[18] *Gartner's 2014 Hype Cycle for Emerging Technologies Maps the Journey to Digital Business*. Aug. 2014. URL: http://www.gartner.com/newsroom/id/2819918 (visited on 03/09/2015) (cit. on p. 1).

[19] *Getting Started with Engines*. URL: http://guides.rubyonrails.org/v4.2.0/engines.html (visited on 03/09/2015) (cit. on p. 41).

[GPLv3] *GNU General Public License*. Version 3. Free Software Foundation, June 2007. URL: http://www.gnu.org/licenses/gpl.html (cit. on p. 48).

[20] *Grape — REST-like API micro-framework*. URL: http://intridea.github.io/grape (visited on 03/09/2015) (cit. on p. 3).

[RFC 6570] J. Gregorio et al. *URI Template*. RFC 6570. IETF, Mar. 2012. URL: https://tools.ietf.org/html/rfc6570 (cit. on p. 39).

[core-observe-16] K. Hartke. *Observing Resources in CoAP*. draft-ietf-core-observe-16. IETF, Dec. 2014. URL: https://tools.ietf.org/html/draft-ietf-core-observe-16 (cit. on pp. 3, 4, 6, 9, 14, 16, 19, 21, 30, 38, 41, 42).

[W3C REC eventsource] I. Hickson. *Server-Sent Events*. Recommendation. W3C, Feb. 2015. URL: http://www.w3.org/TR/eventsource (visited on 03/09/2015) (cit. on p. 16).

[IEEE 802.15.4] *IEEE Standard for Local and metropolitan area networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*. IEEE Standard 802.15.4. 2006. URL: http://standards.ieee.org/about/get/802/802.15.html (cit. on p. 13).

[21] S. Jucker. *Securing the Constrained Application Protocol*. Oct. 2012. URL: https://people.inf.ethz.ch/mkovatsc/resources/californium/cf-dtls-thesis.pdf (visited on 03/09/2015) (cit. on p. 7).

[22] D. Kegel. *The C10K problem*. URL: http://www.kegel.com/c10k.html (visited on 03/09/2015) (cit. on p. 6).

[23] S. Keoh et al. *DTLS-based Multicast Security for Low-Power and Lossy Networks (LLNs)*. draft-keoh-dtls-multicast-security-00. IETF, July 2013. URL: https://tools.ietf.org/html/draft-keoh-dtls-multicast-security-00 (cit. on p. 7).

[24] M. Kovatsch. *Scalable Web Technology for the Internet of Things*. URL: http://people.inf.ethz.ch/mkovatsc/resources/mkovatsc-phdthesis-2015.pdf (visited on 03/09/2015) (cit. on pp. 6, 9, 73, 79).

[25] M. Kovatsch, M. Lanter, and Z. Shelby. "Californium: Scalable Cloud Services for the Internet of Things through CoAP". In: *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*. Cambridge, MA, USA, Oct. 2014. URL: http://www.vs.inf.ethz.ch/publ/papers/mkovatsc-2014-iot-californium.pdf (visited on 03/09/2015) (cit. on pp. 6, 20).

[26] 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 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IMIS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 751–756. ISBN: 978-0-7695-4684-1. DOI: 10.1109/IMIS.2012.104. URL: http://dx.doi.org/10.1109/IMIS.2012.104 (cit. on p. 1).

[lwig-coap-01] M. Kovatsch et al. *CoAP Implementation Guidance*. draft-ietf-lwig-coap-01. IETF, July 2014. URL: https://tools.ietf.org/html/draft-ietf-lwig-coap-01 (cit. on pp. 27, 39).

[core-coap-streaming-00] S. Loreto and O. Novo. *CoAP Streaming*. draft-loreto-core-coap-streaming-00. IETF, Mar. 2012. URL: https://tools.ietf.org/html/draft-loreto-core-coap-streaming-00 (cit. on p. 38).

[RFC 6202] S. Loreto et al. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. RFC 6202. IETF, Apr. 2011. URL: https://tools.ietf.org/html/rfc6202 (cit. on p. 16).

[6lo-6lobac-00] K. Lynn et al. *Transmission of IPv6 over MS/TP Networks*. draft-ietf-6lo-6lobac-00. IETF, July 2014. URL: https://tools.ietf.org/html/draft-ietf-6lo-6lobac-00 (cit. on p. 93).

[27] C. Neukirchen. *Introducing Rack*. Feb. 2007. URL: http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html (visited on 03/09/2015) (cit. on p. 3).

[Rack] *Rack interface specification*. URL: http://rubydoc.info/github/rack/rack/master/file/SPEC (visited on 03/09/2015) (cit. on pp. 3, 5, 23–25, 28, 41).

[RFC 7390] A. Rahman and E. Dijk. *Group Communication for the Constrained Application Protocol (CoAP)*. RFC 7390. IETF, Oct. 2014. URL: https://tools.ietf.org/html/rfc7390 (cit. on pp. 3, 14, 21, 39, 56).

[28] *Rails::Railtie*. URL: http://api.rubyonrails.org/classes/Rails/Railtie.html (visited on 03/09/2015) (cit. on pp. 25, 38).

[29] *Reel – Celluloid::IO-powered web server*. URL: https://github.com/celluloid/reel (visited on 03/09/2015) (cit. on pp. 6, 61).

[RFC 6347] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. IETF, Jan. 2012. URL: https://tools.ietf.org/html/rfc6347 (cit. on pp. 4, 7, 21, 78).

[30] *RSpec*. URL: http://rspec.info (visited on 03/09/2015) (cit. on pp. 27, 47, 59).

[31] *Ruby on Rails*. URL: http://rubyonrails.org (visited on 03/09/2015) (cit. on p. 3).

[schmertmann-dice-codtls-01] L. Schmertmann, K. Hartke, and C. Bormann. *CoDTLS: DTLS handshakes over CoAP*. draft-ietf-schmertmann-dice-codtls-01. IETF, Aug. 2014. URL: https://tools.ietf.org/html/draft-ietf-schmertmann-dice-codtls-01 (cit. on p. 7).

[32] D. C. Schmidt. *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. Nov. 1998. URL: https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf (visited on 03/09/2015) (cit. on pp. 6, 9).

[33] D. C. Schmidt et al. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Vol. 2. John Wiley & Sons, 2013 (cit. on pp. 6, 9).

[34] P. Shaughnessy. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. San Francisco, CA, USA: No Starch Press, 2013. ISBN: 1593275277, 9781593275273. URL: http://patshaughnessy.net/ruby-under-a-microscope (visited on 03/09/2015) (cit. on p. 32).

[RFC 6690] Z. Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC 6690. IETF, Aug. 2012. URL: https://tools.ietf.org/html/rfc6690 (cit. on pp. 24, 27, 38, 41, 47, 54).

[core-resource-directory-02] Z. Shelby and C. Bormann. *CoRE Resource Directory*. draft-ietf-core-resource-directory-02. IETF, Nov. 2014. URL: https://tools.ietf.org/html/draft-ietf-core-resource-directory-02 (cit. on pp. 3, 4, 7, 21, 38, 41, 57, 60, 79).

[RFC 7252] Z. Shelby, K. Hartke, and C. Bormann. *Constrained Application Protocol (CoAP)*. RFC 7252. IETF, June 2014. URL: https://tools.ietf.org/html/rfc7252 (cit. on pp. 3, 4, 6, 7, 13, 15, 19, 21, 24, 27–29, 32–34, 36, 38, 39, 41, 49, 53, 56, 67–69, 78).

[35] *Sinatra*. URL: http://www.sinatrarb.com (visited on 03/09/2015) (cit. on p. 3).

[36] *State of the Market: The Internet of Things 2015*. Verizon Communications, Feb. 2015. URL: http://www.verizonenterprise.com/state-of-the-market-internet-of-things (visited on 03/09/2015) (cit. on p. 1).

[37] J. Storimer. *Ruby core classes aren't thread-safe*. URL: http://www.jstorimer.com/pages/ruby-core-classes-arent-thread-safe (visited on 03/09/2015) (cit. on p. 31).

[38] *Thin – A fast and very simple Ruby web server*. URL: http://code.macournoyer.com/thin (visited on 03/09/2015) (cit. on pp. 6, 28).

[dice-profile-09] H. Tschofenig and T. Fossati. *A TLS/DTLS 1.2 Profile for the Internet of Things*. draft-ietf-dice-profile-09. IETF, Jan. 2015. URL: https://tools.ietf.org/html/draft-ietf-dice-profile-09 (cit. on p. 7).

[39] *Unicorn: Rack HTTP server for fast clients and Unix*. URL: http://unicorn.bogomips.org (visited on 03/09/2015) (cit. on p. 6).

Acknowledgements

First of all, I would like to thank my supervisors *Prof. Dr.-Ing. Carsten Bormann*, *Dr.-Ing. Olaf Bergmann*, and *Dipl.-Inf. Florian Junge* for their support during the work on this document. Especially *Carsten Bormann* helped me with numerous suggestions and ideas.

Furthermore, I thank my parents who allowed me to conduct my studies and my girlfriend *Ann-Kathrin Seiz* for her endless support. Special thanks go to *Jorin* for being quite a motivation! *Ingo Becker* helped me with early reviews, discussions and the developer evaluation. *Heiko Westermann* and *Ruben Schuller* reviewed this document. Thanks guys! Special appreciation is also advisable towards *foo@* which was always there for weird leisure time.