ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Lab Project

at the Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Californium

by Daniel Pauli and Dominique Im Obersteg

Spring 2010

ETH student ID:	06-917-769 (Daniel Pauli)
	05-907-506 (Dominique Im Obersteg)
E-mail address:	paulid@student.ethz.ch
	dimobers@student.ethz.ch
Supervisors:	Dipl.-Ing. Matthias Kovatsch
	Prof. Dr. Friedemann Mattern

Date of submission: 2 December 2011

Contents

Chapter 1

Introduction

1.1 The Internet of Things

The development towards the Internet of Things, in which everyday objects become equipped with sensing and actuation capabilities, introduces new requirements concerning the interoperability of embedded system software. Given the heterogeneity in an Internet of Things and the need for flexibility, a scalable application layer is required. In addition, sensor network applications should be convenient to create regardless of the application developer's background. [1] To address the stated challenges, the Internet Engineering Task Force (IETF) proposed Constrained Application Protocol (CoAP) as a possible solution.

1.2 Short Introduction to CoAP

CoAP is a specialized Web transfer protocol for the use with resource constrained networks and nodes. It meets the specialized requirements such as multicast support, low overhead, and the desired level of simplicity. CoAP is designed for machine-to-machine applications and provides built-in resource discovery, an interaction model between application end-points and Web key concepts such as URIs and content negotiation. The ability to translate CoAP easily into HTTP allows for an easy integration with the Web. [2] CoAP supports the transmission of larger amounts of data by splitting the data into blocks for sending and it manages the reassembly on the application layer upon receipt in order to avoid fragmentation on the lower layers. This feature is called 'block-wise transfer' [3]. Furthermore, CoAP allows the observation of resources in a publish/subscribe pattern (observation handling). A server notifies the clients which registered for specific resources upon changes [4].

1.3 Our Work

Californium is our Java implementation of CoAP. With Californium, we present a solution, which is modular and extensible while still providing a level of abstraction that allows a convenient usage of the library. In chapter 4, we provide examples on how to to write a CoAP server or client easily and with only a few lines of code. We demonstrate, how easy it is with our implementation to create new resources and to add them to an existing CoAP server. During the implementation phase the underlying CoAP working draft version increased from 03 to 07 [2], which imposed a challenge in terms of code openness and flexibility.

Chapter 2

Ideas behind Californium

In the following few paragraphs, we would like to point out the fundamental ideas behind Californium that heavily influenced our design decisions.

2.1 Abstraction

Californium aims to shield users from the details of CoAP in order to provide an intuitive, easy-to-use framework to interact with CoAP endpoints or provide specific services. Following that, users do not need to deal with internals like message retransmissions, block-wise transfers and observation handling.

From a Client's view, a RESTful operation can be performed by just creating a GET, POST, PUT, or DELETE request object, specifying the URI of the target endpoint, followed by payload and other options if required. After execution of the request, the resulting responses can be processed either synchronously or asynchronously.

Setting up a CoAP server using Californium just requires to define its resources by providing subclasses that implement the GET, POST, PUT, and/or DELETE request handlers. Thus, no custom message dispatching is required.

Nevertheless, protocol components and properties can still be configured to suit extended requirements.

2.2 Modularity

First, Californium is based on a layered architecture, extending the two-layer approach described in the CoAP draft[2]. This allows an isolated implementation of different aspects such as message retransmission, transactions, and block-wise transfers specified by CoAP over several levels.

Second, message communication is decoupled from the state representation of CoAP endpoints. Although currently not implemented, this design allows the modeling of endpoints similar to remote objects of Java's RMI system. Hence, a client could access resources on a remote endpoint over a local stub, as if it were an object on the local machine.

2.3 Maintainability

As described in 2.2, our implementation is highly modular and therefore also easy to maintain. Future adaptions and changes can be implemented by only changing the corresponding part of the code without going through the hassle of reading thousands of lines of code and changing the architecture.

Protocol constants that may be subject to change in later CoAP versions as well as Californium specific constants, are managed over an easy-accessible object containing library specific parameters that can be synchronized with a configuration file to allow convenient adjustments by users.

2.4 Extensibility

A direct consequence of 2.2 is the augmented extensibility of our CoAP library.
Communication-related features can be added by introducing new layers in the communication stack, which is convenient due to the uniform interfaces of the layers. Adding a new layer provides the possibility to include and exclude certain components for testing purposes. Client- and server-related features can be implemented by extending the Endpoint classes, e.g., to provide proxying support.

2.5 Backwards Compatibility

Californium implements CoAP draft 07 [2]. To ensure that our implementation of CoAP still works with servers and clients based on previous drafts, we also provide a support for reasonable properties of previous drafts, such as option numbers and return codes.

Chapter 3

Architectural Design

3.1 Overview

Figure 3.1 provides an overview over the whole system architecture. The following sections will provide more details about the packages and the classes therein contained.

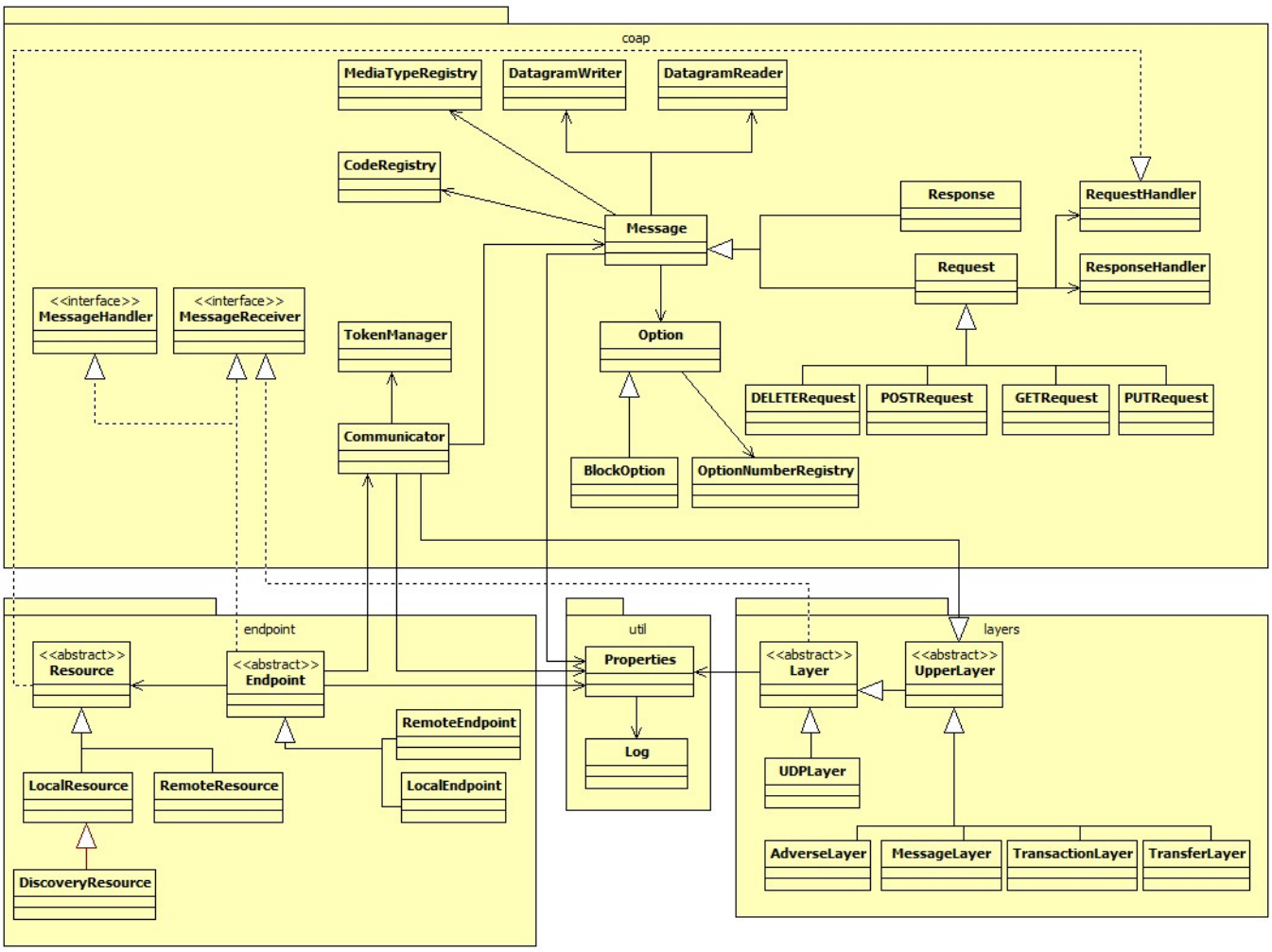

Figure 3.1: Class Diagram

3.2 Packages

3.2.1 Package 'coap'

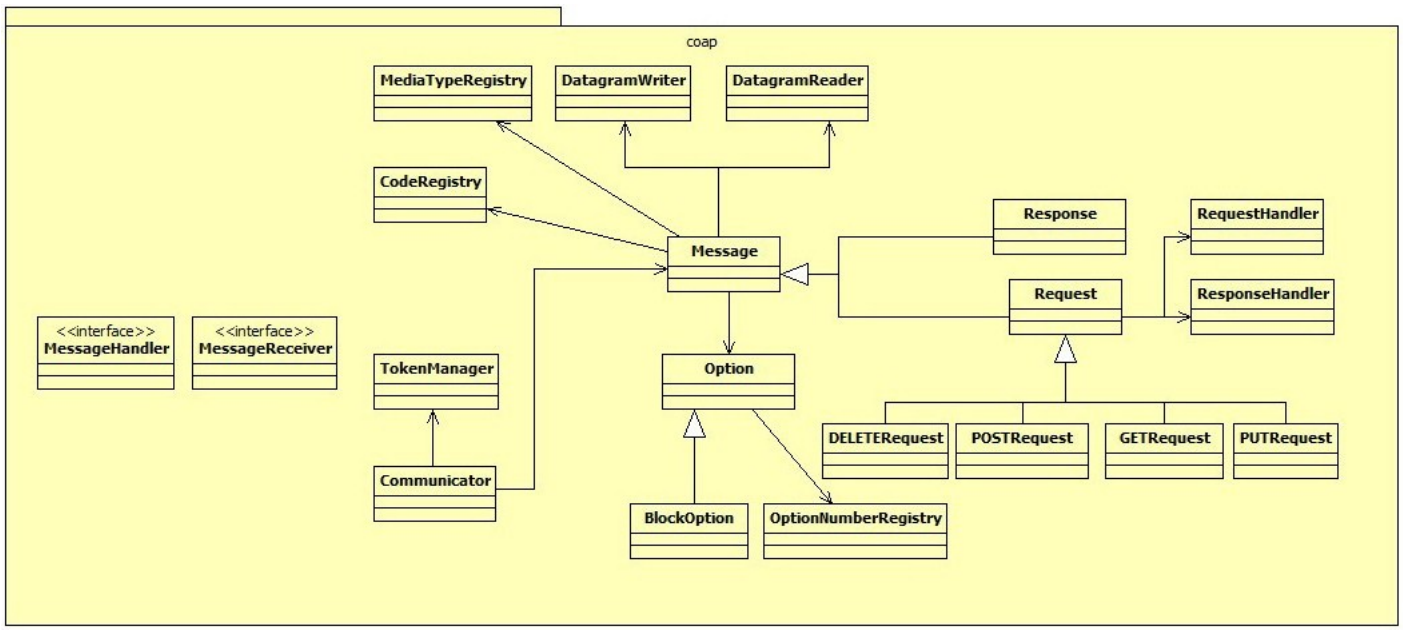

Figure 3.2: Package 'coap'

This package contains all classes that provide the functionality of the core Californium CoAP library. It includes entity classes defining CoAP messages and the derived request and response subclasses as well as static classes containing enumerations and constants for the different registries specified by the IETF CoAP draft [2]. Furthermore, it defines common interfaces that are used to communicate between packages.

Interfaces

MessageHandler
The interface `MessageHandler` describes the method signatures to handle a request or response. It is mainly used by the communication layer classes for the double dispatch to process messages according to their subtype.

MessageReceiver
The interface `MessageReceiver` describes the method signature to receive a message. It is mainly used by communication layer and endpoint classes to implement the propagation of incoming messages.

RequestHandler
The interface `RequestHandler` describes the method signatures to handle the GET, POST, PUT and DELETE request. This interface is implemented by Resource classes and users will implement its methods in order to define custom resources used by Californium Servers.

ResponseHandler
The interface `ResponseHandler` describes the method signature to handle a response. It is used to propagate responses to requests over the affected endpoints back to the communication layer classes.

3.2 Packages

Classes

CodeRegistry
The class `CodeRegistry` provides the CoAP code registry as defined in the draft [2]. It contains constants for request and response codes as well as the corresponding string representations. Users will primarily need this registry to specify or check response codes of CoAP messages.

OptionNumberRegistry
The class `OptionNumberRegistry` provides the CoAP option number registry as defined in the draft[2]. It contains constants for CoAP option numbers as well as the corresponding string representations. Users will primarily need this registry to specify or check additional options of CoAP messages where no convenience accessor/mutator methods (such as Message.setContentType()) are provided.

MediaTypeRegistry
The class `MediaTypeRegistry` provides the CoAP media type registry as defined in the draft [2]. It contains constants for content-type codes as well as the corresponding string representations. Users will primarily need this registry to specify the content-type for custom CoAP messages.

Communicator
The class `Communicator` provides the functionality to build the communication layer stack and to send and receive messages. As a subclass of `UpperLayer` (please refer to the 'UpperLayer' paragraph in section 3.2.3 for a class description) it is actually a composite layer that contains the subsequent layers in the top-down order as explained in table 3.1.

Layer	Description
Transfer	- Support for block-wise transfers using BLOCK1 and BLOCK2 options
Transaction	- Matching of responses to the according requests - Transaction timeouts, e.g., to limit wait time for separate responses and responses to non-confirmable requests
Message	- Reliable transport of Confirmable messages over underlying layers by making use of retransmissions and exponential backoff - Matching of Confirmables to their corresponding Acknowledgment/Reset - Detection and cancellation of duplicate messages - Retransmission of Acknowledgments/Reset messages upon receiving duplicate Confirmable messages
UDP	- UDP datagram exchange

Table 3.1: Californium communication layers

Hence, the Communicator class is used to encapsulate the various communication layers of the CoAP protocol by providing an appropriate unified interface. Internally, it instantiates the required communication layer classes and connects them accordingly. The Communicator also acts as Mediator between endpoint classes and communication layer classes, allowing to specify and query parameters like the UDP port.

Message
The class `Message` represents the core entities for the message exchange between CoAP endpoints. All communication layers process objects of this class. This class provides accessor and mutator methods to the CoAP message header and payload. Internally, it implements the conversion from the Java object to the serialized datagram representation. Users will primarily use the following methods of this class:

```
1 public URI getURI()
2 public void setURI(URI uri)
3 public boolean setURI(String uri)
```

Accesses or modifies the URI property of a message. For outgoing messages, this is the URI of the recipient. For incoming messages, it is the URI of the sender.

```
1 public byte[] getPayload()
2 public String getPayloadString()
3 public void setPayload(byte[] payload)
4 public void setPayload(String payload)
5 public void setPayload(String payload, int mediaType)
```

Accesses or modifies the payload of a message, either as UTF-8 string or raw binary data. The content-type option will be specified accordingly.

```
1 public void addOption(Option opt)
2 public void setOption(Option opt)
3 public void removeOption(int optionNumber)
```

Adds or removes CoAP options from a message. Depending on the option, multiple options with the same option number can be added to a message. addOption() will append an option to the list of options with the same number, setOption() will replace any options with the same number and removeOption() will remove all options with the specified number.

```
1 public List<Option> getOptions(int optionNumber)
2 public void setOptions(int optionNumber, List<Option> opt)
3 public Option getFirstOption(int optionNumber)
```

Accesses or modifies the list of CoAP options with the same option number. getFirstOption() is a convenience method to access the option in case where only one option is allowed, e.g., for the Token option. Note that users will instantiate request and response subclasses rather than directly create a new message object.

Option

The class `Option` provides the functionality of the CoAP options. A message can have several Options with different or same option numbers, as required by the draft. Every option is associated with a value of implicit type and hence provides accessor and mutator methods of different types to the option value. This allows clients to e.g., generate tokens as consecutive integers, while servers will treat token as opaque objects and echo them without further assumptions about their format. Users will primarily use the following methods of this class:

```
1 public Option(int val, int nr)
2 public Option(String str, int nr)
3 public Option(byte[] raw, int nr)
```

Constructor for a new option with the given option number. The option value can be specified either as UTF-8 string, as integer number or as raw binary value, depending on the option number. New options

can be added to messages using the Message.addOption() method.

```
1 public int getIntValue()
2 public void setIntValue(int val)
3 public String getStringValue()
4 public void setStringValue(String str)
5 public byte[] getRawValue()
6 public void setRawValue(byte[] value)
```

Accesses or modifies the value of the option. The option value can be specified either as UTF-8 string, as integer number or as raw binary value, depending on the option number.

BlockOption
The class `BlockOption` provides the CoAP block option as a subclass of `Option`. It is used to encode/decode the NUM, SZX and M fields as well as derived values thereof.

Request
The class `Request` provides the functionality of a CoAP request as a subclass of `Message`. It provides operations to answer a request by a response as well as different ways how to handle incoming responses (by overriding the protected method handleResponse() e.g., using anonymous inner classes, by registering a handler using registerResponseHandler() or by calling the synchronous method receiveResponse()). In order to perform a request, users will instantiate a subclass corresponding to the desired operation (i.e., GET, POST, PUT or DELETE) and use the following methods (in addition to the methods provided by the Message class):

```
1 public void execute() throws IOException
```

Sends the request for processing to the remote endpoint specified by the URI property. This method will return immediately rather than block.

```
1 public void respond(Response response)
2 public void respond(int code)
3 public void respond(int code, String message)
```

Sends a response to the endpoint where the request originated from. This method is generally used in server-side request handlers in order to answer to a processed request. Note that a request can be answered by multiple responses (please refer to the 'Response' paragraph in section 3.2.1 for more information about multiple responses).

```
1 public void accept()
2 public void reject()
```

Accepts or rejects a request by sending an ACK or RST to the endpoint where the request originated from. This methods are generally used in more time-consuming request handlers to acknowledge the receipt of a message immediately, avoiding retransmissions of the request. The final response is then issued using the respond() method, which will be delivered to the sender using a separate message (please refer to the 'Response' paragraph in section 3.2.1 for more information about separate messages). Note that it is not required nor suggested to always perform accept() before respond().

```
1 public void enableResponseQueue(boolean enable)
```

This method is used to define how incoming responses to this request are handled. If the argument is set to true, incoming responses will be queued and can be retrieved using the synchronous receiveResponse() method. If the argument is set to false, incoming responses are not stored and need to be processed using response handlers.

```
1 public Response receiveResponse() throws InterruptedException
```

This method blocks until a response to this request was received. It returns NULL if no response can be received or a timeout occurred. In order to use this method, the response queue must be enabled (see enableResponseQueue() method). This mechanism is generally used by single-threaded clients that only perform one single request at once.

```
1 public void registerResponseHandler(ResponseHandler handler)
2 public void unregisterResponseHandler(ResponseHandler handler)
```

Registers a handler object that is notified about incoming responses to this request. This mechanism is generally used by clients that implement a similar response handling regardless of specific requests, e.g., log the arrival of a response in a file.

```
1 protected void handleResponse(Response response)
```

This protected method is called upon receiving a response to this request. Subclasses can override this method to provide custom response handling. This mechanism is generally used by clients that implement response handling specific to requests, e.g., display the results of a resource discovery. The handler method can be implemented conveniently using anonymous inner classes.

Response

The class `Response` provides the functionality of a CoAP response as a subclass of `Message`. Requests and responses are in a one-to-many-relationship, i.e., a request may receive an arbitrary number of responses (as for separate responses, observations and broadcasts) while a response is related with exactly one request. Users will primarily use the following methods of this class:

```
1 public Request getRequest()
```

Returns the request that is related with this response.

```
1 public int getRTT()
```

Returns the round trip time of this response in milliseconds. The corresponding time stamps are set in the UDP layer.

Requests

The class `GETRequest` provides the functionality of a CoAP GET request message, class `PUTRequest` of a CoAP PUT request message, class `POSTRequest` of a CoAP POST request message, and class `DELETERequest` of a CoAP DELETE request message. All request classes are subclasses of `Request` and they are required by the double dispatch mechanism for the message handling upon receival.

3.2.2 Package 'endpoint'

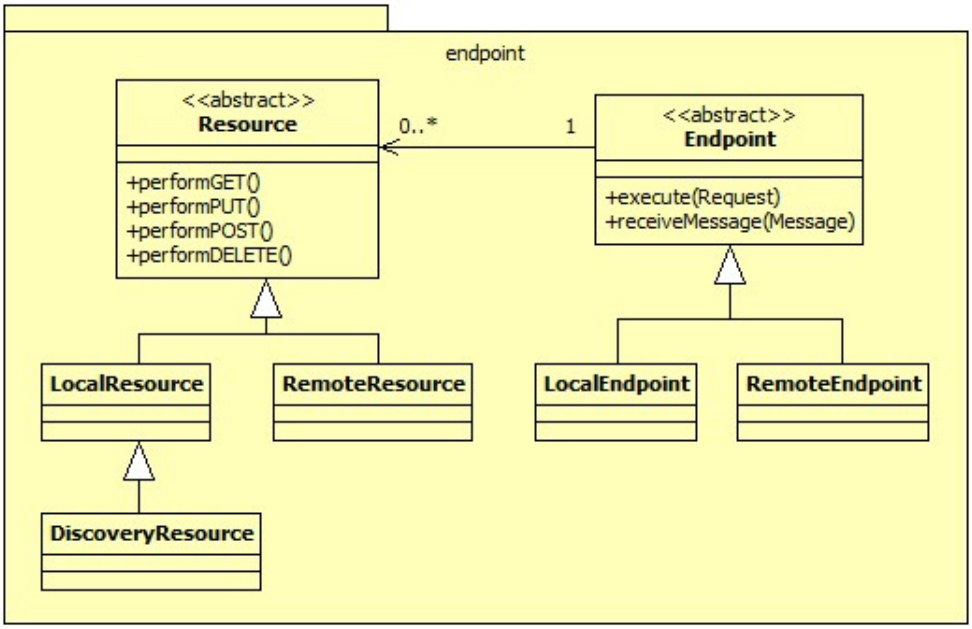

Figure 3.3: Package 'endpoint'

This package contains all classes which provide the functionality for CoAP endpoints, both local and remote. It includes structural classes defining CoAP resources as well as endpoints as generalization of the client-server model. The general idea is to provide a common interface for both local endpoints (corresponding to own server or client implementations using Californium) and remote endpoints (which are only accessible in terms of CoAP message exchanges). Similar as in the Java RMI System, RemoteEndpoint/RemoteResource objects could be implemented as local stubs to LocalEndpoint/LocalResource objects on remote machines, respectively.

Abstract Classes

Endpoint
The abstract class `Endpoint` describes the functionality of a CoAP endpoint as an implementation of the interfaces `MessageReceiver` and `MessageHandler`. It provides an interface to execute requests on a CoAP endpoint as well as access to its resources. An `Endpoint` class provides a `Communicator` object that is used for message exchange. Additionally, it contains a single resource that represents the root of the resource tree.

Resource
The abstract class `Resource` describes the functionality of a CoAP resource as an implementation of the interface RequestHandler. Resources are the core entities of a CoAP endpoint, providing services that can be accessed and manipulated over RESTful operations. Hence, the main purpose of a resource is to process a request and return adequate responses. Resources can contain sub-resources, allowing for content directories or similar, and offer operations to query or iterate over a resource subtree. Finally, as resource can contain several attributes as specified by the CoRE Link Format[5], attributes are represented using a TreeMap in order to support generic attributes, while the resource class also

provides convenient accessor and mutator methods for well-known attributes.

Classes

LocalEndpoint
The class `LocalEndpoint` provides the functionality of a local CoAP endpoint as a subclass of `Endpoint`. A client of the Californium framework will override this class in order to provide custom resources. Internally, the main purpose of this class is to forward received requests to the according resource specified by the Uri-Path option. Furthermore, it implements the root resource to return a brief server description to GET requests with empty Uri-Path.

LocalResource
The class `LocalResource` provides the functionality of a local CoAP resource as a subclass of `Resource`. Users will inherit this class in order to provide custom resources by overriding some the following methods:

```
1 public void performGET(GETRequest request)
2 public void performPOST(POSTRequest request)
3 public void performPUT(PUTRequest request)
4 public void performDELETE(DELETERequest request)
```

These methods are defined by the `RequestHandler` interface and have a default implementation in this class that respond with 4.05 Method Not Allowed.

DiscoveryResource
The class `DiscoveryResource` provides the functionality of a CoAP discovery entry point as a subclass of `LocalResource`. It basically implements the "/.well-known/core" resource and returns the list of resources provided by a CoAP endpoint in link format upon a GET request.

RemoteEndpoint
The class `RemoteEndpoint` provides the functionality of a remote CoAP endpoint as a subclass of `Endpoint`. In future versions, this class may represent a local stub for a remote LocalEndpoint, similar to the Java RMI System.

RemoteResource
The class `RemoteResource` provides the functionality of a remote CoAP resource as a subclass of `Resource`. It is used for resource discovery and in future versions, this class may represent a local stub for a remote `LocalResource`, similar to the Java RMI System.

3.2.3 Package 'layers'

This package contains all classes which provide the functionality required in order to represent and implement the layered architecture of the Californium CoAP design.

Abstract Classes

Layer
The abstract class `Layer` describes a layer in the Californium architecture as an implementation of the interface `MessageReceiver`. It provides features to send and receive CoAP Messages and allows clients

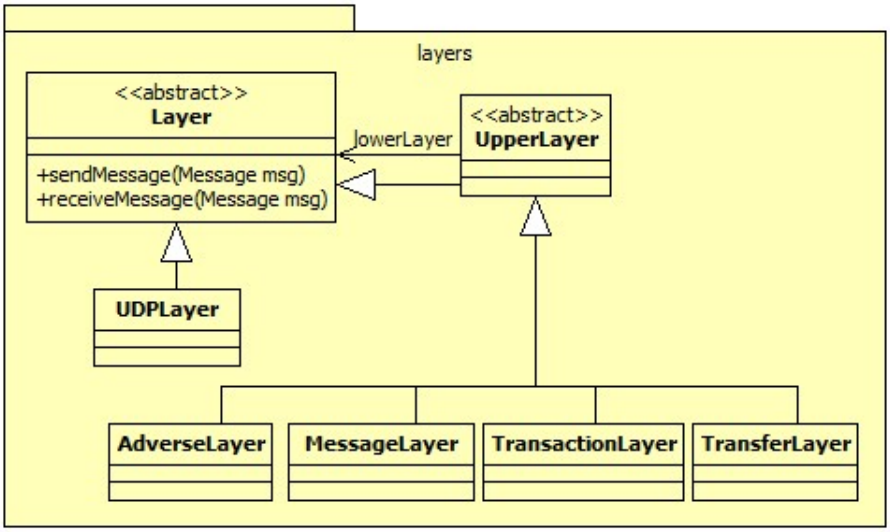

Figure 3.4: Package 'layers'

to subscribe for incoming messages, following the observer pattern. Additionally, it provides an interface for implementing communication statistics.

UpperLayer
The abstract class `UpperLayer` describes a higher communication layer in the Californium architecture as a subclass of `Layer`. Every `UpperLayer` is associated with a underlaying layer instance that is used to send and receive CoAP Messages. All layer classes (except `UDPLayer` as the base layer) inherit from `UpperLayer` and override the protected methods `doSendMessage()` and `doReceiveMessage()` in order to implement additional features to the communication stack. Finally, messages are propagated using methods `sendMessageOverLowerLayer()` and `deliverMessage()`, respectively.

Classes

AdverseLayer
The class `AdverseLayer` provides the functionality of an adverse layer as a subclass of `UpperLayer`. It drops messages with a given probability in order to test retransmissions between `MessageLayer` and `UDPLayer`. Used for testing and evaluation purposes only, it is not present in the final communication stack.

MessageLayer
The class `MessageLayer` provides the functionality of a CoAP message layer as a subclass of `UpperLayer`. It introduces reliable transport of confirmable messages over underlying layers by making use of retransmissions and exponential backoff, matching of confirmables to their corresponding acknowledgment/reset, detection and cancellation of duplicate messages, retransmission of acknowledgments/reset messages upon receiving duplicate confirmable messages.

TransactionLayer
The class `TransactionLayer` provides the functionality of a CoAP transaction layer as a subclass of `UpperLayer`. It implements the matching of responses to the corresponding requests and introduces a

custom timeout for requests to complete.

TransferLayer
The class `TransactionLayer` provides the functionality of a CoAP transfer layer as a subclass of `UpperLayer`. It provides support for block-wise transfers using BLOCK1 and BLOCK2 options and custom block sizes and limits.

UDPLayer
The class `UDPLayer` provides the functionality of a UDP layer that is able to exchange CoAP messages with remote endpoints. According to the UDP protocol, messages are exchanged over an unreliable channel and thus may arrive out of order, appear duplicated or are being lost without any notice.

3.2.4 Package 'util'

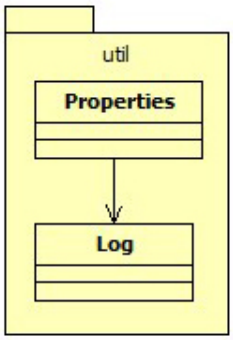

Figure 3.5: Package 'util'

This package contains all classes which provide the functionality required for logging and managing preferences.

Classes

DatagramReader
The class `DatagramReader` provides the functionality to read raw network-ordered datagrams on bit-level. It is used to parse CoAP messages with respect to the integer fields of varying bit lengths.

DatagramWriter
The class `DatagramWriter` provides the functionality to write raw network-ordered datagrams on bit-level. It is used to serialize CoAP messages with respect to the integer fields of varying bit lengths.

Log
The class `Log` provides the functionality to log events in the CoAP library. It is used to redirect console output and provide uniform error messages.

Properties
The class `Properties` implements the functionality of an extended properties registry. It is used to manage CoAP- and Californium-specific constants in a central place.

3.3 Sequences

The following two sequence diagrams show the interaction of the different objects (and layers) when sending (figure 3.6) or receiving (figure 3.7) a message. The implementation details of the operations are split over the different communication layers.

3.3.1 Send

Whenever an endpoint calls execute on a request, the send order is propagated down the different layers until it reaches the UDPLayer which then sends the datagram over UDP.

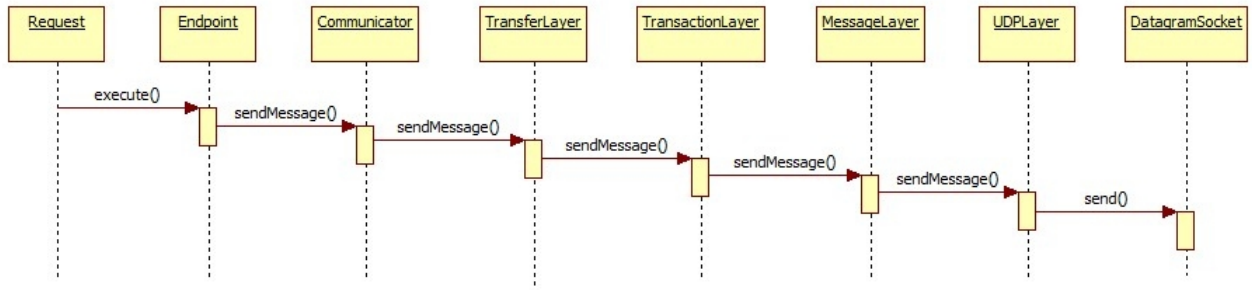

Figure 3.6: Send operation

3.3.2 Receive

Whenever the UDPLayer receives a message (response to a request), the receive order is propagated up the different layer until it reaches the Endpoint which then handles the response.

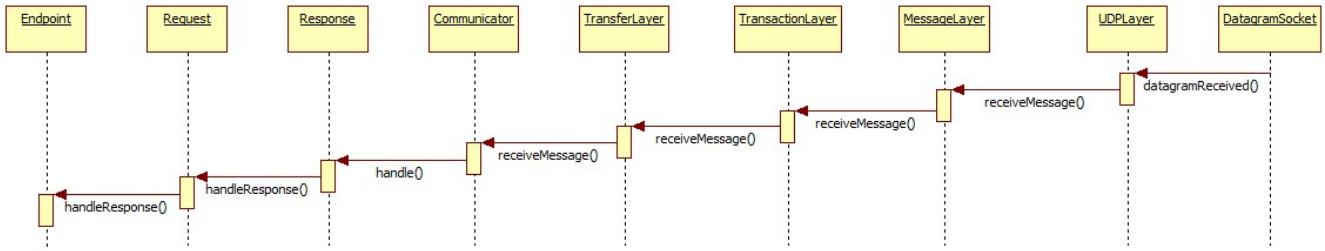

Figure 3.7: Receive operation

3.3.3 Response Timeout

Whenever the UDPLayer does not receives a response to a request, i.e., a receive time out has occurred, it repeatedly sends the request again until a response is received.

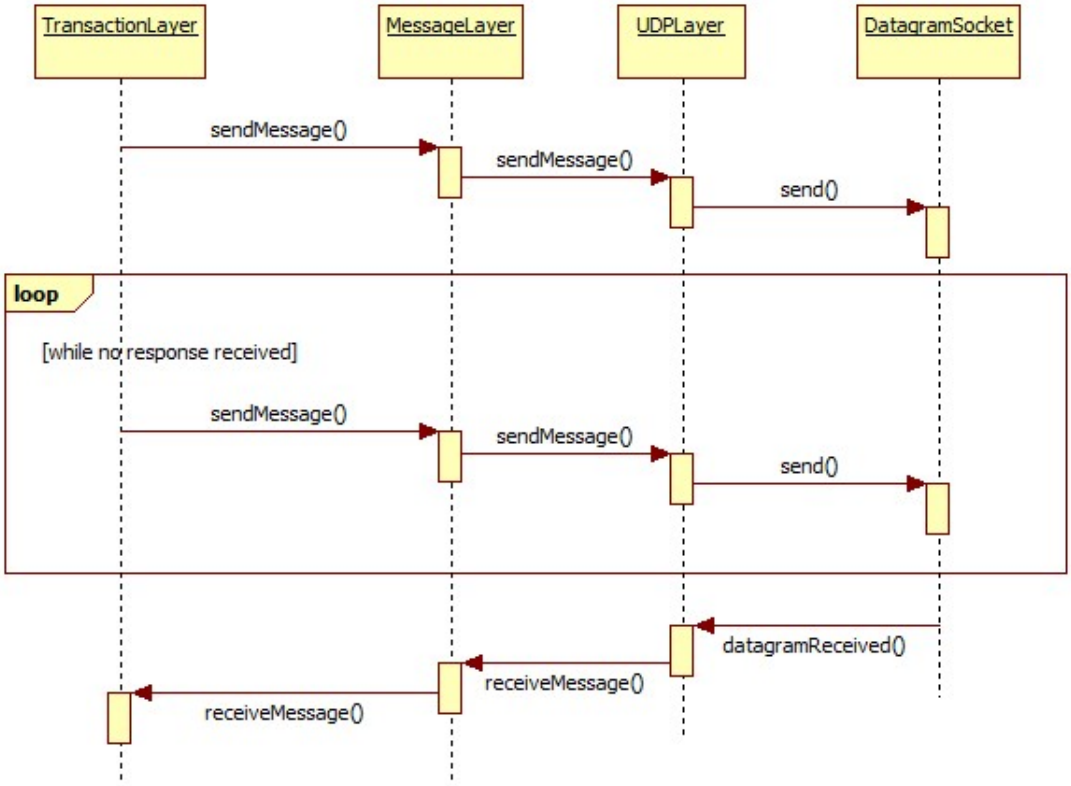

Figure 3.8: Response time out

Chapter 4

Examples

4.1 Clients

The code below shows different ways of how to perform a request and handle responses.

4.1.1 Synchronous receive

This example demonstrates the receiveResponse() method. This approach is generally used by single-threaded clients that only perform one single request at once, e.g. console applications.

```
1  public class GETClient {
2
3    public static void main(String args[]) {
4
5      URI uri = null; // URI parameter of the request
6
7      if (args.length > 0) {
8
9        // input URI from command line arguments
10       try {
11         uri = new URI(args[0]);
12       } catch (URISyntaxException e) {
13         System.err.println("Invalid URI: " + e.getMessage());
14         System.exit(-1);
15       }
16
17       // create new request
18       Request request = new GETRequest();
19       // specify URI of target endpoint
20       request.setURI(uri);
21       // enable response queue for synchronous I/O
22       request.enableResponseQueue(true);
23
24       // execute the request
25       try {
26         request.execute();
27       } catch (IOException e) {
28         System.err.println("Failed to execute request: " + e.getMessage());
29         System.exit(-1);
30       }
31
```

```
32        // receive response
33        try {
34          Response response = request.receiveResponse();
35
36          if (response != null) {
37            // response received, output a pretty-print of it
38            response.log();
39          } else {
40            // transaction timeout occurred
41            System.out.println("No response received.");
42          }
43
44        } catch (InterruptedException e) {
45          System.err.println("Receiving of response interrupted: " + e.getMessage
                ());
46          System.exit(-1);
47        }
48
49      } else {
50        // display help
51        System.out.println("Usage: GETClient URI");
52      }
53    }
54  }
```

4.1.2 Using listener

This example demonstrates how to register a client class to handle responses. This approach is generally used by clients that implement a similar response handling regardless of specific requests, e.g. log the arrival of a response in a file.

```
1  static class MyClient implements ResponseHandler {
2
3    /*
4     * Implement your client functionality here
5     *
6     */
7
8    public void performSampleRequest() {
9
10     // create new request
11     Request request = new GETRequest();
12
13     // specify URI of target endpoint
14     request.setURI("coap://vs0.inf.ethz.ch:5683/timeResource");
15
16     // register client as response handler
17     request.registerResponseHandler(this);
18
19     // execute the request
20     try {
21       request.execute();
22     } catch (IOException e) {
23       System.err.println("Failed to execute request: " + e.getMessage());
24       System.exit(-1);
```

```
25      }
26
27      /*
28       * Do something or return to the message loop of the GUI
29       */
30    }
31
32    @Override
33    public void handleResponse(Response response) {
34      // response received, output a pretty-print
35      response.log();
36    }
37  }
```

4.1.3 Subclassing

This example demonstrates how to use an anonymous inner class to handle responses. This approach is generally used by clients that implement response handling specific to requests, e.g. display the results of a resource discovery.

```
1  public void performSampleRequest() {
2
3    // create new request using an anonymous inner class
4    Request request = new GETRequest() {
5      @Override
6      protected void handleResponse(Response response) {
7        // response received, output a pretty-print
8        response.log();
9      }
10   };
11
12   // specify URI of target endpoint
13   request.setURI("coap://vs0.inf.ethz.ch:5683/timeResource");
14
15   // execute the request
16   try {
17     request.execute();
18   } catch (IOException e) {
19     System.err.println("Failed to execute request: " + e.getMessage());
20     System.exit(-1);
21   }
22
23   /*
24    * Do something or return to message loop
25    */
26 }
```

4.2 Hello World Server

The code below demonstrates how to implement a simple CoAP server that provides a resource which returns "Hello World" upon a GET-Request.

```
1  public class HelloWorldServer extends LocalEndpoint {
2
3    class HelloWorldResource extends ReadOnlyResource {
4
5      /*
6       * Constructor for a new Hello-World resource. Here,
7       * resource-specific properties are set.
8       */
9      public HelloWorldResource() {
10
11       // call constructor of superclass and
12       // provide resource identifier
13       super("helloWorld");
14
15       // set display name
16       setResourceTitle("Hello-World Resource");
17     }
18
19     /*
20      * Implementation of the actual functionality by overriding
21      * the default GET request handler
22      */
23     @Override
24     public void performGET(GETRequest request) {
25       // respond to the request with the according response code and payload
26       request.respond(CodeRegistry.RESP_CONTENT, "Hello World!");
27     }
28   }
29
30   /*
31    * Constructor for a new Hello-World server. Here,
32    * the resources of the server are initialized.
33    */
34   public HelloWorldServer() throws SocketException {
35     // provide an instance of the Hello-World resource
36     addResource(new HelloWorldResource());
37   }
38
39   /*
40    * Application entry point.
41    */
42   public static void main(String[] args) {
43
44     try {
45       // create server
46       HelloWorldServer server = new HelloWorldServer();
47       System.out.println("Server listening on port " + server.port());
48
49     } catch (SocketException e) {
50       System.err.println("Failed to initialize server: " + e.getMessage());
51     }
```

```
52    }
53  }
```

4.3 Storage Resource

The code below demonstrates a more complex resource of a server that can be used to store hierarchical data like a file system.

Content of the individual resources can be accessed using GET/PUT requests, new sub-resources of given Uri-Paths can be created using POST requests and finally be removed using DELETE requests. The resources can be observed.

```
1  public class StorageResource extends LocalResource {
2
3    /*
4     * Default constructor.
5     */
6    public StorageResource() {
7      this("storage");
8    }
9
10   /*
11    * Constructs a new storage resource with the given resourceIdentifier.
12    */
13   public StorageResource(String resourceIdentifier) {
14     super(resourceIdentifier);
15     setResourceTitle("PUT your data here or POST new resources!");
16     setResourceType("Storage");
17     setObservable(true);
18   }
19
20   // REST Operations ////////////////////////////////////////////////////
21
22   /*
23    * GETs the content of this storage resource.
24    * If the content-type of the request is set to application/link-format
25    * or if the resource does not store any data, the contained sub-resources
26    * are returned in link format.
27    */
28   @Override
29   public void performGET(GETRequest request) {
30
31     // create response
32     Response response = new Response(CodeRegistry.RESP_CONTENT);
33
34     // check if link format requested
35     if (request.hasFormat(MediaTypeRegistry.LINK_FORMAT) || data == null) {
36
37       // respond with list of sub-resources in link format
38       response.setPayload(toLinkFormat(), MediaTypeRegistry.LINK_FORMAT);
39
40     } else {
41
42       // load data into payload
```

```
43        response.setPayload(data);
44
45        // set content type
46        response.setContentType(getContentTypeCode());
47      }
48
49      // complete the request
50      request.respond(response);
51    }
52
53    /*
54     * PUTs content to this resource.
55     */
56    @Override
57    public void performPUT(PUTRequest request) {
58
59      // store payload
60      storeData(request);
61
62      // complete the request
63      request.respond(CodeRegistry.RESP_CHANGED);
64    }
65
66    /*
67     * POSTs a new sub-resource to this resource.
68     * The name of the new sub-resource is retrieved from the request
69     * payload.
70     */
71    @Override
72    public void performPOST(POSTRequest request) {
73
74      // get request payload as a string
75      String payload = request.getPayloadString();
76
77      // check if valid Uri-Path specified
78      if (payload != null && !payload.isEmpty()) {
79
80        createNew(request, payload);
81
82      } else {
83        // complete the request by reporting error
84        request.respond(CodeRegistry.RESP_BAD_REQUEST,
85          "Payload must contain Uri-Path for new sub-resource.");
86      }
87    }
88
89    /*
90     * Creates a new sub-resource with the given identifier in this
91     * resource, recursively creating sub-resources along the Uri-Path
92     * if necessary.
93     */
94    @Override
95    public void createNew(Request request, String newIdentifier) {
96
97      // omit leading and trailing slashes
98      if (newIdentifier.startsWith("/")) {
```

```
99        newIdentifier = newIdentifier.substring(1);
100     }
101     if (newIdentifier.endsWith("/")) {
102       newIdentifier = newIdentifier.substring(0, newIdentifier.length()-1);
103     }
104
105     // check if resource should be created as a sub-resource
106     // of the current (this) resource
107     int delim = newIdentifier.indexOf('/');
108     if (delim < 0) {
109
110       // create new sub-resource if it does not yet exist
111       StorageResource resource = (StorageResource)subResource(newIdentifier);
112       if (resource == null) {
113
114         // create new resource and add it to the current resource
115         resource = new StorageResource(newIdentifier);
116         addSubResource(resource);
117
118         // store payload
119         resource.storeData(request);
120
121         // create new response
122         Response response = new Response(CodeRegistry.RESP_CREATED);
123
124         // inform client about the location of the new resource
125         response.setLocationPath(resource.getResourcePath());
126
127         // complete the request
128         request.respond(response);
129
130       } else {
131
132         // resource already exists; update data
133         storeData(request);
134
135         // complete the request
136         request.respond(CodeRegistry.RESP_CHANGED);
137       }
138
139     } else {
140
141       // split path in parent and sub identifier
142       String parentIdentifier = newIdentifier.substring(0, delim);
143       newIdentifier = newIdentifier.substring(delim+1);
144
145       // retrieve corresponding sub-resource, create if necessary
146       StorageResource sub = (StorageResource) subResource(parentIdentifier);
147       if (sub == null) {
148         sub = new StorageResource(parentIdentifier);
149         addSubResource(sub);
150       }
151
152       // delegate creation to the sub-resource
153       sub.createNew(request, newIdentifier);
154     }
```

```
155   }
156
157   /*
158    * DELETEs this storage resource, if it is not root.
159    */
160   @Override
161   public void performDELETE(DELETERequest request) {
162
163     // disallow to remove the root "storage" resource
164     if (parent instanceof StorageResource) {
165
166       // remove this resource
167       remove();
168
169       request.respond(CodeRegistry.RESP_DELETED);
170     } else {
171       request.respond(CodeRegistry.RESP_FORBIDDEN,
172         "Root storage resource cannot be deleted");
173     }
174   }
175
176   // Internal ////////////////////////////////////////////////////////////
177
178   /*
179    * Convenience function to store data contained in a
180    * PUT/POST-Request. Notifies observing endpoints about
181    * the change of its contents.
182    */
183   private void storeData(Request request) {
184
185     // set payload and content type
186     data = request.getPayload();
187     setContentTypeCode(request.getContentType());
188
189     // signal that resource state changed
190     changed();
191   }
192
193   private byte[] data;
194 }
```

Chapter 5

Future Work

As CoAP is still in Internet-Draft status, Californium is a work in progress as well. In this section, we suggest further features that may be implemented to further increase the capabilities of Californium.

5.1 Block Size Negotiation

So far, the transfer layer performs block-wise transfer if requested from the sender-side or if the message to send is too large. However, the layer accepts messages of all possible sizes and does not negotiate a certain block size. In a future version of Californium, this feature will have to be added for draft compliance.

5.2 Streaming-based Transfers

Currently, block-wise transfer is implemented transparent to the user in a way such that endpoints will only receive messages with complete payload. More precisely, partial incoming messages are stored at the TransferLayer and delivered only upon completion of the transfer such that, upon processing of the received message, the entire payload is available. This design could be improved by introducing a streaming-oriented interface for accessing message payload. Like this, a message handler can start processing the payload right at the arrival of the first message block by reading from the stream, blocking if the data is not yet available. This technique also conforms to standard Java I/O libraries where streams are used to read from sockets and files. Core implementational change to incorporate this feature may be to replace the byte array storing the payload of a message with a pair of connected PipedOutput/InputStreams that basically allow one thread to write in a buffer while a second thread reads and processes the buffer content, blocking if necessary.

5.3 Address Representation

In the current implementation, a message stores only one URI (for outgoing messages the address of the recipient and for incoming messages the address of the sender). This address is internally stored as instance of the java.net.URI class which holds the path to a certain destination. This representation does not precisely match all requirements that may arise from further use cases of Californium, e.g., when dealing with proxying and it is also not optimal with respect to separation of semantically different information. Therefore, a message will need to store the address and port of the source and the destination in order to avoid a context dependent message URI. Furthermore, URI-host, URI-path and proxy information will need to be stored separately to avoid a combination of information which is semantically different. [6]

5.4 CoAP Endpoint Framework

The goal of Californium is to support remote endpoints and resources as local stubs analog to java's remote method invocation, however the `RemoteEndpoint` and `RemoteResource` classes do not yet hold the corresponding remote context. This feature is added best, when a Californium client application has been created which offers complete ability to represent and modify arbitrary remote resources. The key functionality can then be refactored into the two classes.

5.5 Security

The current Californium implementation does not take security issues into account. To mitigate the risk of security threats, adequate countermeasures will have to be implemented in a future version.

Bibliography

[1] M. Kovatsch, "Firm Firmware and Apps for the Internet of Things," in *Proceedings of the 2nd ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA 2011)*, Honolulu, HI, USA, May 2011.

[2] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," http://tools.ietf.org/html/draft-ietf-core-coap-07, 2011.

[3] C. Bormann and Z. Shelby, "Blockwise transfers in CoAP," http://tools.ietf.org/html/draft-ietf-core-block-03, 2011.

[4] K. Hartke and Z. Shelby, "Observing Resources in CoAP," http://tools.ietf.org/html/draft-ietf-core-observe-02, 2011.

[5] Z. Shelby, K. Hartke, and C. B. and B. Frank, "CoRE Link Format," http://tools.ietf.org/html/draft-ietf-core-link-format-06, 2011.

[6] Matthias Kovatsch, E-Mail, 2011.