UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA, ELETTRONICA E DELLE TELECOMUNICAZIONI

PROGETTAZIONE E SVILUPPO DI UN CROSS-PROXY PER L'INTEROPERABILITÀ TRA PROTOCOLLI APPLICATIVI IN SCENARI DI INTERNET OF THINGS

DESIGN AND IMPLEMENTATION OF A CROSS-PROXY FOR THE INTEROPERABILITY AMONG APPLICATION-LAYER PROTOCOLS IN INTERNET OF THINGS SCENARIOS

Relatore:

Chiar.mo Prof. Ing. Simone Cirani

Correlatore:

Dott. Ing. Marco Picone

Tesi di Laurea di:

ERIC CAMELLINI

ANNO ACCADEMICO 2013/2014

Dedico questa tesi al WASN Lab e in particolare ai Prof. Simone Cirani e Marco Picone, che mi hanno seguito durante tutta l'attività, che hanno valorizzato quel che ho sviluppato, integrandolo in progetti dell'intero laboratorio, e che mi hanno trasmesso la passione per gli ambiti trattati.

Contents

Chapter 1

Introduction

1.1 IP-based Internet of Things

The Internet of Things (IoT) can be defined as the network of physical objects accessed through the Internet [1]. The concept was defined in early 2000s and it is simple but powerful: if all objects in daily life were equipped with identifiers and wireless connectivity, these objects could communicate with each other and be managed by computers. At the time, this vision required major technology improvements because there were many problems related to the connection of mobile objects or sensors, such as their reduced battery life, their wireless connection issues, the costs, and the limited range of addresses.

Figure 1.1: A possible IoT Scenario.

Today, many of these issues have been solved, the size and cost of wireless radios has dropped dramatically and, due to the adoption of IPv6, it is possible to address billions of devices. The IoT now describes a system where items in the physical world (and sensors within or attached to these items) are connected to the Internet in a wireless or wired way, as shown in Figure 1.1. These sensors can use various types of local area connections (RFID, NFC, Wi-Fi, Bluetooth, or IEEE 802.15.4) and can also have cellular connectivity (GSM, GPRS, 3G, or LTE).

1.2 Low-power and lossy networks

Low-power and Lossy networks (LLNs) are made up of many embedded devices with limited power, memory, and processing resources. They can be interconnected by a variety of protocols, such as IEEE 802.15.4, Bluetooth and Low Power WiFi. This thesis is focused on an applicaion-layer protocol suited to communication in LLNs: the Constrained Application Protocol (CoAP) and on its related protocol translation issues, detailed in Sections 1.2.3, 2.2 and Chapter 3, respectively.

1.2.1 6LowPAN

IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [2] is an Internet Engineering Task Force (IETF) working group (WG), chartered to bring IPv6 communication over LLNs. The idea is that the Internet Protocol could and should be applied even to the smallest devices and that low-power devices with limited processing capabilities should be able to participate in the Internet of Things. Following this aim, the 6LoWPAN group has defined encapsulation and header compression mechanisms that allow IPv6 packets to be sent to and received from IEEE 802.15.4 based network.

Other IETF WGs are working on other areas related to 6LoWPAN: for instance, the Routing Over Low power and Lossy networks Working Group (ROLL) WG has created a routing protocol for constrained node networks called RPL, and the CoRE WG has defined CoAP.

1.2.2 RPL

IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [3] is a routing protocol designed for LLNs, which provides mechanisms to perform point-to-point, multipoint-to-point, and point-to-multipoint traffic. This protocol is useful because such networks typically feature traffic patterns that may not always be point-to-point and that may potentially comprise up to thousands of nodes; these characteristics offer unique challenges to a routing solution. The ROLL WG defined application-specific routing requirements for LLN routing protocol and RPL was designed from the WG itself with the objective to meet them.

1.2.3 CoAP

The Constrained Application Protocol (CoAP) [4] is a specialized application-layer transfer protocol, created as a lightweight version of the HyperText Transfer Protocol (HTTP), to be used with constrained nodes and networks. The nodes often have small amounts of ROM and RAM and limited microcontrollers (8-bit), while constrained networks such as 6LoWPAN often have a limited throughput (tens of kbit/s) and a high packet error rate. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation: it provides a request/response interaction model between application endpoints, supports the service/resource discovery in a built-in way, and includes key concepts of the Web such as URIs and Internet media types for the content.

CoAP has been designed by the Constrained RESTful Environments (CoRE) WG and one of its objectives is to easily interface with HTTP, and so be able to integrate with the Internet, while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments. For the integration needed in a IoT scenario, HTTP/CoAP protocol translation is necessary to communicate between LLNs with CoAP and the Internet, as shown in Figure 1.2. The construction of this translator component is a major achievement of this thesis, which is detailed in Section 1.4.

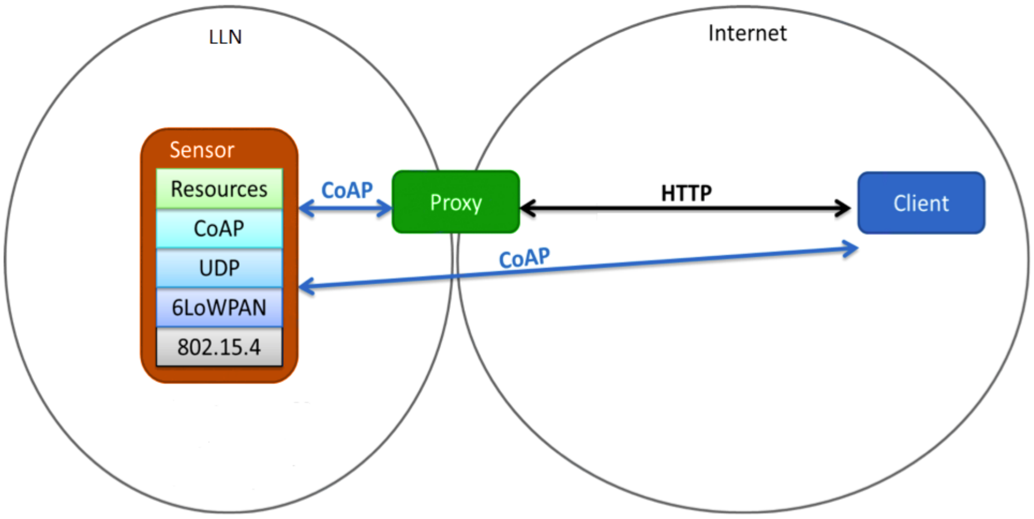

Figure 1.2: A CoAP scenario and its interaction with the Internet.

1.3 IoT scenarios and Smart-X applications

The first application of IoT was to create a communication network between industrial equipment, but in these days many more application have been found. Cisco created a new definition that summarizes the objectives of the last researches in this branch: *Internet of Everything*, that is, a network that could contain industrial items but also everyday objects and living beings such as animals, to monitor and track them, or plants. This new vision applied to industries or cities is the base for the concepts described in Sections 1.3.1 and 1.3.2.

1.3.1 Smart Manufacturing

The basis for recent industrial improvement is the automatic communication between products, systems, and machines. With IoT and machine-to-machine (M2M) communication, it is possible to foresee scenarios where sensors collect data and share them with other industrial equipment, relying on intelligent networks along the entire value chain that communicate and control each other autonomously, with significantly reduced intervention by operators.

Stefan Ferber, Director for business development of the IoT at Bosch Software Innovations, said: “Industry 1.0 was the invention of mechanical help, Industry 2.0 was mass production, pioneered by Henry Ford, Industry 3.0 brought electronics and control systems to the shop floor, and Industry 4.0 is peer-to-peer communication between products, systems and machines.” [5].

1.3.2 Smart Cities

A Smart City [6] is the city of the future, where IoT technologies are used to connect things and people, according to their needs. It’s easy to find many applications of IoT in urban scenarios, such as the optimization of services related to transportation (e.g., traffic management, parking, and transit systems). Singapore has adopted an intelligent transport strategy and set of systems that made it one of the least congested major cities. It has an express way monitoring and advisory system that alert motorists, a GPS system installed on the city taxis, which monitors and reports traffic conditions, and a Control Centre, which consolidates the data and provides real-time traffic informations to the public.

Another possible application are smart parkings: wireless sensors embedded in parking spots that can gather their real-time status tracking if a spot is occupied or empty, and send this information to a data management system, linked to servers and/or mobile applications, which can be used by drivers. These examples of IoT applications in urban scenarios show how the smart city approach could bring positive societal impact, such as the minimization of traffic congestion, reduction of carbon emissions and elimination of inefficiencies associated with parking enforcement.

1.4 Thesis objectives

The main objective of this thesis is to develop a Cross-Proxy implementing HTTP-to-CoAP protocol translation with the support the mjCoAP library (detailed in Section 2.3.1), allowing a constrained network to be accessed from the Internet with the most used application-layer protocol (i.e., HTTP). This part is explained in Chapter 3.

Another objective, together with the development of the Cross-Proxy, is to extend that mjCoAP library with other external packages implementing some of the CoAP extra functions described in other drafts defined by the IETF CoRE Working Group. In particular, this work is focused on the Observing and the Blockwise Transfer functions (detailed in Chapter 4), and also on the improvement of the mjCoAP library's main features.

Chapter 2

State of the art

In this chapter, an overview of IoT-related work is presented, focusing on the CoAP protocol and its implementation. The IETF CoRE working group has designed CoAP with the aim at realizing the *REST* architecture in a suitable form for constrained nodes.

2.1 RESTful architecture and applications

The REpresentational State Transfer (REST) [7] is an architectural model intended to easily create, read, update or delete informations on a server using simple HTTP calls. It is an alternative to more complex mechanisms like Simple Object Access Protocol (SOAP), Common Object Request Broker Architecture (CORBA) and Remote Procedure Call (RPC). A REST call is simply an HTTP request to a server. CoAP has the same call structure because it is based on this architectural style. More generally, REST is a software architectural style consisting of a coordinated set of constraints applied to components, connectors, and data elements within a distributed hypermedia system.

The REST architectural style was developed by World Wide Web Consortium Technical Architecture Group (W3C TAG) in parallel with HTTP 1.1 and the World Wide Web represents the largest implementation of a system conforming to the REST architectural style. The REST architecture involves three kind of elements:

- *Components* provide the data through an interface;
- *Connectors* connect the components;
- *Data* are transferred between components using the connectors.

The architectural properties of REST are realized by applying some interaction constrains to the mentioned elements, so that the creation of new REST based protocols and extensions would not violate them, because they are what makes the web successful. Some of these restrictions that the components must conform to are:

- **Client - Server model**: a server owns the data and a client can request these data, providing them to the user through an interface. In some cases the client/user can modify these data using that interface.
- **Stateless model**: a request contains everything that is needed to provide a response, with no need to save the client's state on the server.
- **Caching support**: a client can cache responses saving them on its internal memory.
- **Layered structure**: a client cannot know if it is connected to an intermediary server or to an endpoint of the network.
- **Code on demand support**: a server can transfer code to clients temporary extending their functionalities.

REST is also applied to services and resources provided by a server by using service APIs that adhere to the REST constraints and are called RESTful. RESTful APIs are defined with a base Uniform Resource Identifier (URI) [8], which is a string used to identify a resource, like http://example.com/resources/resource1, an Internet media type for the data (e.g., JSON) and standard HTTP methods (verbs) to interact with these resources (GET, PUT, POST, or DELETE).

The standard syntax of a URI is shown in Figure 2.1.

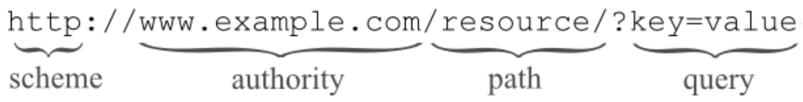

Figure 2.1: URI Syntax; in detail, the authority can be divided in host and port where the host can be a name or an IP address (www.example.com:8080 or 192.168.0.1:8080). The path section could contain more elements divided by a "/" and the same is for the query section, where the elements are separated by the character "&".

The REST HTTP standard methods are the same used in CoAP and are the way to access the resources. These methods are listed below and an example of a REST communication is shown in Figure 2.2:

- **GET**: used to retrieve the content/state of a resource;
- **PUT**: used to modify or update a resource;
- **POST**: works like PUT method, but allows to create a new resource if it doesn't exists;
- **DELETE**: used to delete a resource.

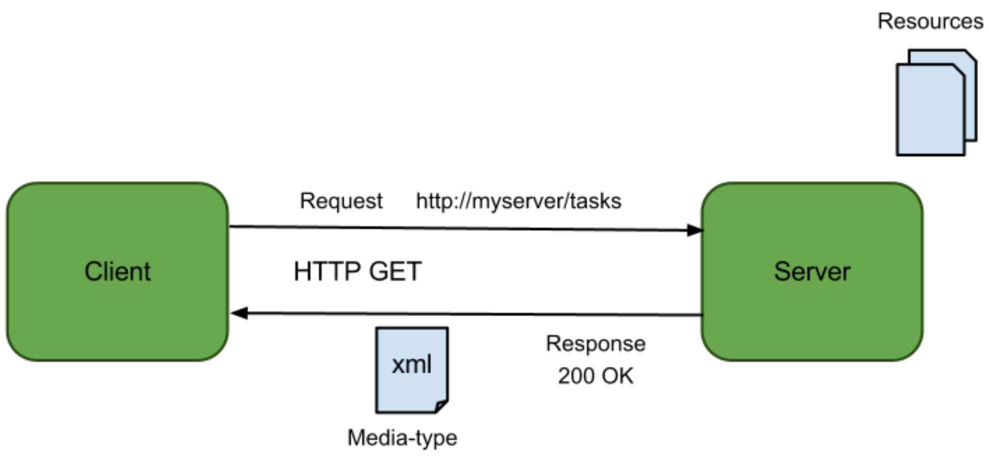

Figure 2.2: A REST Client - Server communication.

2.2 CoAP protocol

The Constrained Application Protocol [4] has been designed to provide a realization of the REST architecture in a suitable form for constrained nodes: the goal of CoAP is not to simply compress HTTP (which is too heavy for constrained devices), but rather to realize a subset of REST common with HTTP and optimized for M2M applications in LLNs, with related specific functions, like the multicast support. Designed by the IETF CoRE Working Group, the final version of the CoAP draft (version 18) has just become an IETF standard (RFC 7252). Like HTTP, CoAP is an application-layer protocol and must run on top of a transport-layer protocol: it uses UDP so it shares with it the maximum packet size, the unreliability problems, and the absence of a pre-established connection during the transmission.

Unlike HTTP (which is a text-based protocol) CoAP is a UDP-based binary protocol that sends messages under the form of byte arrays and, like HTTP, can access to a resource through a request, that must be sent to the resource URI: a CoAP URI has the standard syntax, the URI scheme must be *coap://* and the default port of the protocol is 5683. The request possible methods are the ones listed before (GET, POST, PUT and DELETE, as detailed in Section 2.1) and they operate the same way HTTP does.

Another similarity with HTTP is that a response is returned to the client after a request, and contains a code that indicates the result of the attempt to understand and satisfy it: this code uses the format $c.dd$, where c is the class (1-5), and dd is the detail as a two-digit decimal (for instance, 4.04 Not Found, the class is 4 that stands for "client error").

2.2.1 CoAP messaging

The CoAP messaging model is based on the exchange of messages (byte arrays) over UDP between endpoints, with optional reliability. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options (similar to the HTTP headers) and a payload (content). This message format is shared by requests and responses and each message contains a Message ID, used to detect duplicates, and a token used to match responses to requests.

There are four message types: Confirmable (used to provide reliability), Non-confirmable, Acknowledgement and Reset. If a message is confirmable (CON) the recipient must respond with an acknowledgement (ACK) that has the same message ID, or with a reset (RST) if it's not able to respond. If no ACK response is returned prior to a retransmission timeout (with exponential back-off), the message is retransmitted. A message that does not require reliable transmission, for example each single measurement out of a stream of sensor data, can be sent as

Non-confirmable (NON) and it will not be followed by an ACK (the RST is optional also, in case of inability to process it).

Another important mechanism is the *piggy-backed response*: if a message is CON the response could be sent from the recipient after the acknowledgement, in another message, or in the same message. In the last case that we mentioned, the response will be an ACK with also a response code and an optional payload, and it's called piggy-backed response. An example is shown in Figure 2.3.

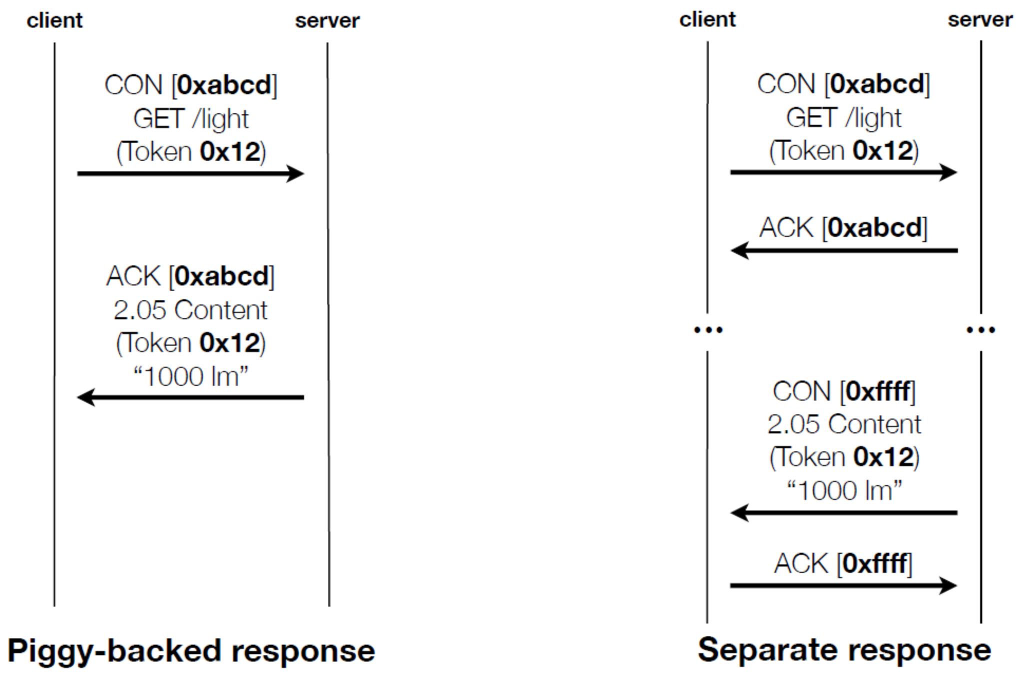

Figure 2.3: CoAP communication with separated and piggy-backed response.

Another important part of CoAP messages is *options*. Options have the role to add informations to a message, for example the media type of the payload or what media type will be accepted in response. Each option instance specifies the Option Number of the defined CoAP option, the length of the Option Value and the Option Value itself. It must be noticed that in CoAP the path and query sections of the

URI are split in their components and sent in a series of URI-path and URI-query options.

Finally, it is also worth mentioning that a secure version of CoAP exists: DTLS-secured CoAP. Just like HTTP is secured using Transport Layer Security (TLS) over TCP (and the URI scheme is *https://*), CoAP is secured using Datagram TLS (DTLS) and the URI scheme *coaps://*.

2.2.2 CoAP extensions

CoAP has also been extended with some other related drafts that can be found on the CoRE WG page of the IETF website (http://tools.ietf.org/wg/core/). The most prominent drafts are:

- **Observe**: with this extension a client can perform a registration to a resource through a GET request with the new CoAP Observe option, and then it'll receive notification messages with the new values when this resource changes [9];
- **Blockwise Transfer**: using this kind of transfer allow to split a CoAP request or response in a certain number of chunks that will be sent as different CoAP messages. It is useful because of the limited memory of constrained nodes, that sometimes can't handle big payloads, or if the message reaches the UDP maximum packet size [10];
- **HTTP mapping**: this draft's objective is to define the rules for the HTTP to CoAP protocol translation and vice versa, along with the guidelines for the development and the placing of a proxy used for that scope [11].

This detailed functionalities and issues related to their implementation will be explained in Chapter 3 and Chapter 4.

2.3 CoAP related work

2.3.1 mjCoAP

MjCoAP is an open-source implementation of the CoAP protocol based on Java and developed by the University of Parma. It supports the final version the CoAP draft, therefore RFC 7252, and provides a set of classes to develop Java applications for CoAP messaging. The most relevant classes and interfaces provided by the library are the following:

- *CoapMessage*: a generic CoAP message;
- *CoapProvider*: a class for sending and receiving messages, used with a CoapProviderListener that intercepts them;
- *CoapTransactionServer*: a class that defines methods to send responses intercepted by a CoapTransactionServerListener;
- *CoapTransactionClient*: a class that defines methods to send requests and to receive the related responses through a CoapTransactionClientListener;
- *Utilities*: there are many other classes made to implement other CoAP aspects or to create objects like CoapMessageFactory, CoapOption, CoapMethodCode.

A simple CoAP client with a method that can send a GET request is shown in Listing 2.1, while a simple CoAP server listening for GET and POST requests, but that handles only the former, is shown in Listing 2.2.

```java
public class CoAPClient{
    CoapProvider coapProvider;
    public CoAPClient(int coapPort) throws IOException {
        this.coapProvider = new CoapProvider(coapPort);

    public void sendPOSTMessage(String payload, SocketAddress destAddr){
        CoapMessage coapRequestMessage = CoapMessageFactory.createRequest(false,
            CoapMethodCode.POST, payload.getBytes());
        new CoapTransactionClient(this.coapProvider,destAddr,this).request(
            coapRequestMessage);
        //using "this" as parameter for the listener means that this class will receive
            the response
    }
    //To catch the responses this class must implement the CoapTransactionClientListener
         or the CoapProviderListener interfaces
}
```

Listing 2.1: mjCoAP client example.

```java
public class SimpleCoAPServer implements CoapProviderListener{
    CoapProvider coapProvider;
    public SimpleCoAPServer(int coapPort){
        try {
            this.coapProvider = new CoapProvider(coapPort);
            this.coapProvider.addListener(CoapMessageSelector.messageMethodSelector(
                CoapMethodCode.GET),this);
            this.coapProvider.addListener(CoapMessageSelector.messageMethodSelector(
                CoapMethodCode.POST),this);
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onReceivedMessage(CoapProvider coap_provider, CoapMessage msg) {
        if(msg.getCode()==CoapMethodCode.GET){
            String response = "{\"value\":\"test\"}";
            new CoapTransactionServer(coap_provider,msg,listener).respond(
                CoapResponseCode.responseCode(2,5),response.getBytes());
        }
        else{
            //Method not allowed 4.05
            new CoapTransactionServer(coap_provider,msg,listener).respond(
                CoapResponseCode.responseCode(4,5),null);
        }
    }
}
```

Listing 2.2: mjCoAP server example.

2.3.2 Other CoAP libraries

Along with mjCoAP, there are other relevant projects that implement CoAP. The most prominent are:

- *Califorium (Cf)* is another open source Java based library to support the CoAP communication, developed by a research lab. of the ETH Zurich (Swiss Federal Institute of Technology);
- *Libcoap* is a library for the C language CoAP support, the one the TinyOS CoAP implementation is based on (Tiny OS is an operating system for constrained nodes like Contiki OS – see Section 2.3.4);
- *Erbium (Er)* (see Section 2.3.4) is a C-based library, official CoAP Implementation for Contiki OS, and it has been developed by the same team that created Californium.

2.3.3 Copper (Cu)

Copper (Cu)[1] is a Firefox extension that installs a handler for the *coap* URI scheme and allows users to browse and interact with IoT devices. Developed by the same

[1]https://addons.mozilla.org/it/firefox/addon/copper-270430/

team from that released Californium and Erbium (see Section 2.3.2), it supports interaction through GET, POST, PUT, and DELETE, handles the automatic retransmission in case of unreceived ACK, and many other functionalities. It also implements some extensions mentioned in Section 2.2.2 like Blockwise Transfer, resource observing and the related notification receiving. It is really useful for testing CoAP applications.

2.3.4 Contiki OS

Contiki is an open source operating system for networked, memory-constrained systems with a particular focus on low-power wireless IoT devices. Contiki was created by Adam Dunkels at the Swedish Institute of Computer Science (SICS) in 2002 and has been further developed by a world-wide community of people and organizations. Even though it provides complex features, such as multitasking and a built-in full protocol stack, it only needs a few kilobytes of memory: for instance, a full system, complete with a graphical user interface, uses about 30 kilobytes of RAM. The system supports the recently standardized IETF protocols for low-power IPv6 networking, including the 6LowPAN adaptation layer, the RPL IPv6 routing protocol, and the CoAP RESTful application-layer protocol.

Support for CoAP is possible with the Erbium (Er) library, a low-power REST Engine that includes a comprehensive embedded CoAP implementation, which became the official one for Contiki OS. It supports draft-ietf-core-coap-03, draft-ietf-core-coap-12, draft-ietf-core-coap-13, and draft-ietf-core-coap-18, together with Blockwise Transfer and observing function (see Section 2.2.2). Another important feature is the possibility to use Cooja, the Contiki network simulator, which makes developing and debugging tremendously easier. It provides a simulation environment that allows developers to both see their applications run in large-scale networks or in extreme detail on fully emulated hardware devices. [12]

2.4 Other protocols and utilities

CoAP is not the only application-layer protocol that can be used for the smart-object communication. For instance, HTTP that is based on text (i.e. on transferring sequences characters) and is the standard for Internet navigation. Another application-layer protocol is BitTorrent, used to transfer large amounts of data on a peer-to-peer communication and often used in file sharing. Other protocols also based on different architectures (not RESTful) may also be used.

2.4.1 Dev HTTP Client (DHC)

DHC is a Google Chrome Extension, which implements a REST HTTP Client. It has been used in the thesis as a HTTP client for testing of the HTTP-to-CoAP Proxy. Using DHC as a testing client allows to select all the message headers, the content and everything else is needed.

2.4.2 Jetty HTTP server

The HTTP-to-CoAP Cross-Proxy acts on the HTTP side as an HTTP server that will receive the messages to be translated. Jetty is a pure Java based HTTP (Web) server developed by the Eclipse Foundation. It can handle different requests simultaneously, in multi-threading, and allows to set a fully customizable request handler.

Chapter 3

HTTP-to-CoAP Cross-Proxy

CoAP has been designed with the objective to be an application-layer protocol specialized for constrained environments and to be easily used in REST architectures such as the Web. One of the related issues is that CoAP should be able to easily interoperate with HTTP through an intermediary proxy which performs cross-protocol conversion, so that the devices on the Internet could communicate with a CoAP node directly with the constrained protocol, if supported, or via HTTP through the proxy, as shown in Figure 3.1.

One of the main issues in the development of this network element is *protocol translation*: an HTTP message addressed to a constrained node must pass through the proxy, which converts it to a CoAP message with the same options (if compatible) and characteristics. The proxy then waits for the response from the node and returns it to the HTTP client after another conversion, this time from CoAP to HTTP. Another problem is *addressing*, because the message must be sent to the proxy with an HTTP URI, but should point to a CoAP resource, whose URI should be contained in the HTTP URI.

In this chapter, these issues and the solutions chosen during the development of the Java based HTTP-to-CoAP (HC) Cross-Proxy are discussed. In order to guarantee full interoperability, the IETF CoRE WG guidelines of the http-mapping draft [11] have been followed.

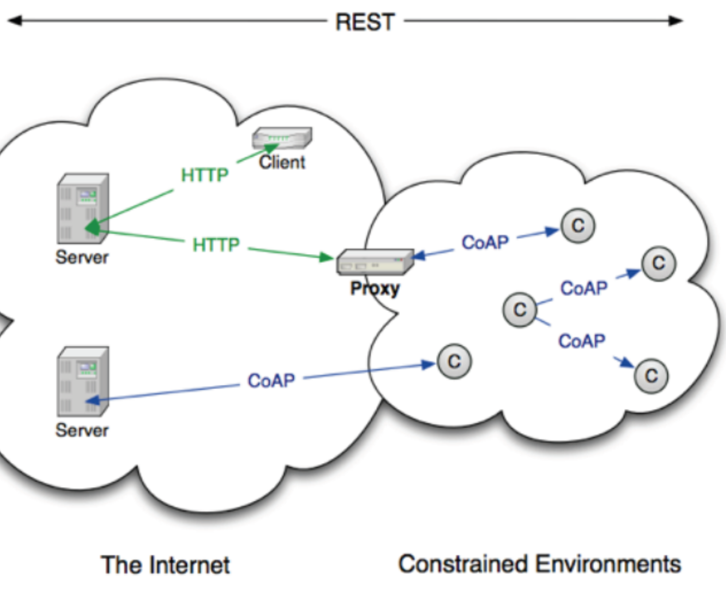

Figure 3.1: A CoAP network with a Cross-Proxy.

3.1 Use cases examples

To illustrate in which situations HTTP-to-CoAP request mapping may be used, some use cases are briefly described:

- *Smartphone and sensors*: A smartphone, when in the same network of a sensor, can perform CoAP requests directly to it. When the smartphone is at a remote location, the same request could be done by an authenticated "https" request from the smartphone over an external IP network (the Internet) through an HTTP-CoAP proxy;

- *Making sensor data available*: An HTTP-to-CoAP proxy could be configured to expose sensor data to the world via the web (HTTP and/or HTTPS). The sensor can only handle CoAP requests, and data are exposed by sending HTTP requests to the proxy.

3.2 Forward Proxy and Reverse proxy

It is possible to implement two kinds of proxy:

- **Forward Proxy**: a message forwarding agent that is selected by the client, usually via local configuration rules, to receive requests of absolute URI and to attempt to satisfy them via protocol-translation, where the protocol is indicated in the URI. In this case the user decides to use the proxy for a predefined subset of the URI space.

- **Reverse Proxy**: a receiving agent that translates the received requests to the underlying server's protocol. It behaves as an origin (HTTP) server on its connection towards the (HTTP) client and as a (CoAP) client on its connection towards the (CoAP) origin server.

Forward and Reverse proxies are very similar: the main difference is the former is set as a client configuration setting, that addresses the messages through it, while when using the latter, the client is unaware it is communicating with a proxy, that so it is seen as a server (HTTP) that publishes the resources. In this work, a reverse proxy has been implemented.

3.3 URI mapping

URIs are formed of different components, as described in Section 2.1, and it is important to know that the URI scheme does not imply that a particular protocol is used to access the service, so it is possible to define the same resource to be accessible by different protocols. The problem is that HTTP clients typically only support *http://* and *https://* schemes, so they cannot directly access CoAP servers (which support *coap://* and/or *coaps://*). To solve that issue the client must “pack” the CoAP URI into an HTTP URI so that it can be (non-destructively) transported from the user agent to the HC Proxy that can then “unpack” the CoAP URI and finally de-reference it via a CoAP request to the target node.

3.3.1 Possible mapping templates

There are different choices for the packing of the request URI, and all of them require the base URI of the proxy. So it’s important to say that the Cross-Proxy must be started with a pre-defined base URI. Below are listed (as examples) the possible URI encapsulations, using the base URI *http://p.example.com/hc*:

- Default with the CoAP URI in the path component:
  http://p.example.com/hc/coap://s.example.com/light?dim=5

- Default with optional scheme (without ’coap://’):
  http://p.example.com/hc/s.coap.example.com/foo

- As query element with ’coap’ scheme):
  http://p.example.com/hc?coap_target_uri=coap://s.example.com/light

- As query element with ’coaps’ scheme):
  http://p.example.com/hc?coaps_target_uri=coaps://s.example.com/light

- As query element with 'coap' or 'coaps' scheme:
  http://p.example.com/hc?target_ uri=coaps://s.example.com/light
  http://p.example.com/hc?target_ uri=coap://s.example.com/light

- Enhanced form with CoAP URI components in path segments and optional query in query component:
  http://p.example.com/hc/coap/s.example.com/light?on

- Enhanced form with CoAP URI components split in individual query arguments:
  http://p.example.com/hc?s=coap& hp=s.example.com& p=/light& q
  http://p.example.com/hc?s=coaps& hp=s.example.com& p=/light& q=on

When the authority of the target CoAP URI is given as an IPv6 address, the surrounding square brackets must be percent-encoded in the hosting HTTP URI, in order to comply with the syntax defined in URI RFC for a URI path segment, and also the % symbol must percent encoded if present in an IPv6 address. Everything else can be safely copied from the Target CoAP URI to the Hosting HTTP URI. For example:

*coap://**[**2001:d**%**b8::1**]**/light?on*

becomes

*http://p.example.com/hc/coap://**%5B**2001:d**%25**b8::1**%5D**/light?on.*

All these templates of mapping are supported by the proxy developed in this thesis, along with the discovery function that allow to fetch informations on these possible URI packing methods from the HTTP side by sending the following request to the proxy:

GET http://p.example.com/.well-known/core?rt=core.hc

3.4 Protocol translation

In our reverse proxy scenario a client sends an HTTP request to the proxy with the encapsulated CoAP URI that must be extracted (see Section 3.3). The HTTP message must then be translated in a CoAP message and the proxy, like a CoAP client, sends it to the node addressed by the extracted URI. In the developed proxy this CoAP request is implemented as a blocking one, so the message handler waits for the response from the CoAP node and converts it into an HTTP response. This response is then retuned to the first client: an example is shown in Figure 3.2).

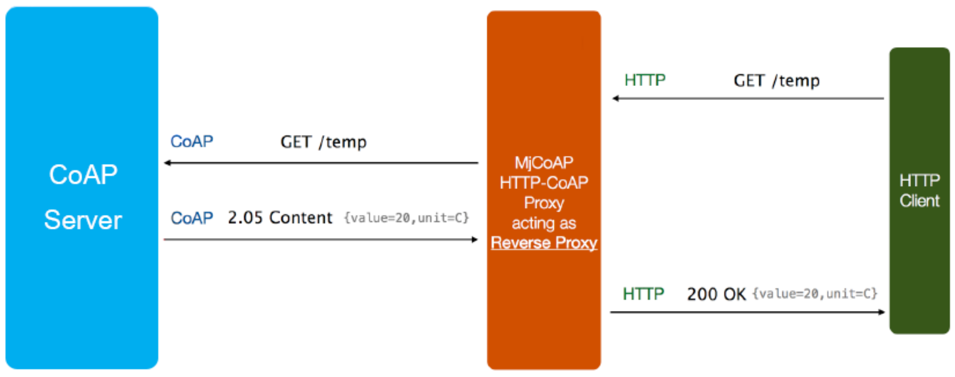
CoAP Server
CoAP
GET /temp
CoAP 2.05 Content {value=20,unit=C}
MjCoAP HTTP-CoAP Proxy acting as Reverse Proxy
HTTP
GET /temp
HTTP 200 OK {value=20,unit=C}
HTTP Client

Figure 3.2: The flow of an HTTP request that passes through the proxy.

All this steps are transparent to the client, which may assume it is communicating with the intended target HTTP server that exposes the resources directly, without retrieving them from nodes of a CoAP sub-network. Because of that the proxy must be placed at the edge of the constrained network, running on a component that also provides an HTTP interface to the external network.

Another function that can be implemented on the proxy is *caching* of the CoAP responses from the CoAP nodes. In this way, the proxy would be able to directly handle an HTTP request.

3.4.1 HTTP-to-CoAP request translation

The translation of a request consists in different parts that must be analysed and converted from HTTP to CoAP. The main steps, which have been implemented, are:

1. **Method translation**: it's easy to extract the CoAP equivalent of the HTTP request's method since, as described in Section 2.1, the standard methods are the same (i.e., GET, POST, PUT, and DELETE). If the proxy receives an HTTP request with a method that CoAP doesn't support (e.g. OPTIONS, HEAD, TRACE or CONNECT) the response will be *501 Not Implemented*.

2. **URI mapping**: the URI is extracted if received in the form of one of the templates described in Section 3.3. The CoAP node address is extracted from authority and the various URI-path and URI-query option chunks to be inflated in the translated request are extracted from the path and query parts. If the URI encapsulation is not correct a *400 Bad Request* response is sent to the client.

3. **Headers to Options**: HTTP provides a large number of Headers for a message, but only the ones that have a CoAP related Option can be translated. Other options are discarded and only the compatible ones are converted (i.e., *Accept*, *Content Type*, *If Match*, *If None Match* and *ETag*; *Content Type* is translated as the Content-Format Option while the others have the same name).

4. **Media type translation**: Another issue of the protocol translation is that HTTP supports hundreds of media types while CoAP only six of them in the standard version. Those that can be mapped from HTTP are *text/-plain*, *application/link-format*, *application/xml*, *application/octet-stream*, *application/json*, and *application/exi*, all mapped as the one with the same name for CoAP. If the HTTP media type is not supported by CoAP the message cannot be translated and a *415 - Unsupported Media Type* is sent in response.

5. **Payload copying**: If the Content Length HTTP header contains a value that is not zero then there is a not empty body in the message, so it must be copied in the CoAP message's payload after its conversion into a byte array.

Note that some of these passages are not mentioned or clarified in the http-mapping draft; in fact, these are choices made for the sake of implementation of the Cross-Proxy.

3.4.2 CoAP-to-HTTP response translation

The translation of the response related to the CoAP request, previously converted from the HTTP one, is a procedure similar to the one described in Section 3.4.1. The main steps are:

1. **Response code translation**: the response is returned with a code that must be mapped in the related HTTP one. Some codes are the same (e.g *2.01 Created* mapped as *201 Created*) but for many of them a direct mapping is not possible. Table 3.1 shows the CoAP-to-HTTP response code mapping.

2. **Options to Headers**: this procedure is the opposite of the one described in Section 3.4.1. The CoAP Options that can be translated into HTTP Headers

are:

- *Content-Format*: mapped as the Content Type header;
- *Location-Path and Location-Query*: the chunks of this options, that represents the location (path and query) of a created resource, are joined and inflated in the Content Location Header;
- *Etag*: translated as the Etag header, same name in this case;
- *Max-Age*: this last option that can be returned in the response is mapped as the Cache Control header with the value "max-age=A" where A is the Integer value of the option converted to String. A particular case is when a response with code *5.03* is returned (see Table 3.1);

3. **Media type translation**: In that case there are no problems in the media type mapping because all the CoAP standard formats that the response could contain are translatable in HTTP, so they are mapped (As described in the previous procedure, but in the opposite way) when the *Content-Format* option is translated in the *Content Type* header;

4. **Payload copying**: if the CoAP message contains a payload, this is copied in the HTTP message body as a string built from the byte array.

3.5 Implementation

The proxy implementation is based on the mjCoAP library (see Section 2.3.1), and on three main components: the HTTP Server, the Translator, and the CoAP Client, which are described next.

3.5.1 The HTTP Server

The Cross-Proxy HTTP Server is the main class, *HttpToCoapProxy*, and was realized by using Jetty HTTP Server 2.4.2 set up with a proper request handler. The message handler is a class called *HttpRequestHandler* that extends the class *AbstractHandler* of the Jetty library, and its code executes whenever an HTTP request arrives and stops when the related response is sent. Jetty handles automatically the multi-threading so when a message arrives, the handler manages it also if another message is being served. The handler follows the next steps:

1. **Initial checks**: in this step the handler checks if the request method and media-types can be translated into the CoAP related ones, by calling static methods from the helper class *HttpToCoapMapper*. If the message doesn't pass through the checks a proper error is sent in response: *501 Not Implemented* or *415 Unsupported Media Type*.

2. **Request translation**: if the message passes the checks it can be translated to a CoAP request using the Translator.

3. **Blocking request**: the CoAP request obtained, after translation, is sent to the proper address through the CoAP Client by calling a blocking method that puts the server in a waiting status. When a CoAP response is returned the execution restarts.

4. **Response translation**: to translate the CoAP response to an HTTP one a Translator's method is called.

5. **Response sending**: the HTTP final response is sent to the client.

If the final response returned by the CoAP Client is null a *404 Not Found* error is sent: that's an implementation choice not mentioned in the draft.

3.5.2 The Translator

The *Translator* class implements the *ITranslator* interface that contains two methods: the first returns a CoAP request from an HTTP one passed as a parameter, and the second does the opposite translation with the response. An interface is used so that different implementations of the *ITranslator* can be set for the message handler. The implementation translates the messages by calling methods from the helper classes described in Section 3.5.4.

3.5.3 The CoAP Client

For the CoAP side, a *BlockwiseCoapClient*, described in Section 4.1.5, is used. It's a blocking client as requested by the handler, its *sendMessage()* method sends the CoAP request and returns the related response or a *null* value if no reply arrives in 29 second (Timeout). This client also handles the Blockwise Transfer as described in Section 4.1. Proxy's Timeout and Blockwise Transfer support are also mentioned in the http-mapping draft [11] as optional features that the can be implemented.

3.5.4 Other components

The implementation of the *ITranslator* interface is based on four static classes: *HttpToCoapMapper*, *CoapToHttpMapper*, *HttpToCoapUriMapper*, and *WellKnownCoreHc*. The former two handle the translation as described in Section 3.4 and the latter two manage the URI mapping with discovery as described in Section 3.3.

3.6 Using the Cross-Proxy

The code to start the HTTP-to-CoAP Cross-Proxy is shown in Listing 3.1. It is also possible to set the parameters for the Blockwise Transfer (see Section 4.1) with setter methods of the *HttpRequestHandler*; If not set, the default values are used (implementation choice).

```java
int httpPort = 8080; //Default http port
int coapPort = 5683;

try{
        CoapProvider coapProvider = new CoapProvider(coapPort);
        String basePath = "/hc";
        HttpRequestHandler httpHandler = new HttpRequestHandler(basePath,coapProvider,
            new TranslatorDraft03());
        HttpToCoapProxy proxy = new HttpToCoapProxy(httpPort,httpHandler);
        proxy.start();
    }catch (SocketException e){
        e.printStackTrace();
    }
```

Listing 3.1: HTTP-to-CoAP Proxy launcher example.

CoAP code	**HTTP code**
2.01 Created	201 Created
2.02 Deleted	200 OK 204 No Content *HTTP code is 200 or 204 respectively in the case that a CoAP server returns a payload or not.*
2.03 Valid	304 Not Modified 200 OK *Depending on the caching options, not implemented in the described proxy*
2.04 Changed	200 OK 204 No Content *Like for the 2.02, see above*
2.05 Content	200 OK
4.00 Bad Request	400 Bad Request
4.01 Unauthorized	400 Bad Request
4.02 Bad Option	400 Bad Request
4.03 Forbidden	403 Forbidden
4.04 Not Found	404 Not Found
4.05 Method Not Allowed	400 Bad Request
4.06 Not Acceptable	406 Not Acceptable
4.12 Precondition Failed	412 Precondition Failed
4.13 Request Entity Too Large	413 Request Repr. Too Large
4.15 Unsupported Media Type	415 Unsupported Media Type
5.00 Internal Server Error	500 Internal Server Error
5.01 Not Implemented	501 Not Implemented
5.02 Bad Gateway	502 Bad Gateway
5.03 Service Unavailable	503 Service Unavailable *The value of the HTTP "Retry-After" response-header is taken from the value of the CoAP Max-Age Option, if present.*
5.04 Gateway Timeout	504 Gateway Timeout
5.05 Proxying Not Supported	502 Bad Gateway

Table 3.1: Response codes mapping table.

Chapter 4

CoAP Extensions

In this chapter, the extra CoAP functionalities implemented, as specified in the drafts on the IETF CoRE Working Group website (http://tools.ietf.org/wg/core/), and the work done for the improvement of the mjCoAP library are described.

4.1 Blockwise Transfer

The Blockwise Transfer is a CoAP extra function, described in the related draft [10], that allows the protocol to split a message in a certain number of chunks. The request or responses are chunked in the following cases:

1. An arriving message is too big to be handled by the node, for example because of the size of the buffer where the incoming ones are stored;
2. A message is too big to be sent, for example because of his payload size.

The size limit is called Blockwise Threshold and can be different for every implementation but the message, with all the header, cannot be larger than the UDP maximum packet size (i.e. 65507 Bytes).

To implement that feature two new options are defined in the draft: *Block1* and *Block2*, that are used respectively in a chunked request or response. For every chunk of the message all the options are the same, but the Block ones change to keep track of the number of blocks already sent or received. Both these options have the size of a byte that contains three parameters:

- *NUM* (bit 0-3): contains the number of the block that is being requested or provided;
- *M* (bit 4): contains flag for control usage, it's meaning is different for *Block1* and *Block2*;
- *SZX* (bit 5-7): contains an integer value that indicates the size of the block and it's calculated using the following formula: $BLOCKSIZE = 2^{SZX+4}$.

4.1.1 Block1 example

The *Block1*1 option (see Figure 4.1) is used when the client sends a POST or a PUT message with the payload split in chunks: every chunk contains a *Block1*1 option with NUM indicating the sent block's number, M = 1 if the block is not the last one and SZX calculated with the proper formula (see above in this section). For every chunk received the server sends an ACK with the same block option, or with a smaller value in SZX indicating that the following blocks must be sent with a smaller size, with a consequent NUM value change. That mechanism is called size negotiation and is also shown in Figure 4.1. A case where a client must begin a Blockwise Transfer, if supported, is when after a try with the full message, a *4.13 Request Entity Too Large* error is returned (it is a new code defined in the draft). If the block doesn't arrive in order or some of them are lost (and also all their retransmissions) a *4.08 Request Entity Incomplete* is returned (another new code).

4.1.2 Block2 example

The *Block2* option (see Figure 4.2) is used when the client sends a message (GET) and the response is slit in chunks: every chunk is an ACK with payload (a piggy-backed response, see Section 2.2) and contains a *Block2* option with NUM indicating the block number, M = 1 if the block is not the last one and SZX calculated with the proper formula (see above in this section). For every chunk received the client sends a request for the further one, with M = 0 and the same SZX. The request of a block could contain a smaller SZX value indicating that the following blocks must be sent with a smaller size, with a consequent NUM value change in the request where SZX changes (late negotiation, shown also in Figure 4.2). If the client knows that the response will be chunked, it could include a *Block2* option with values 0:0:SZX in the first request, where SZX indicate the preferred block size (early negotiation). *Block1* and *Block2* combination is possible: a request sent in chunks could be answered with a chunked response.

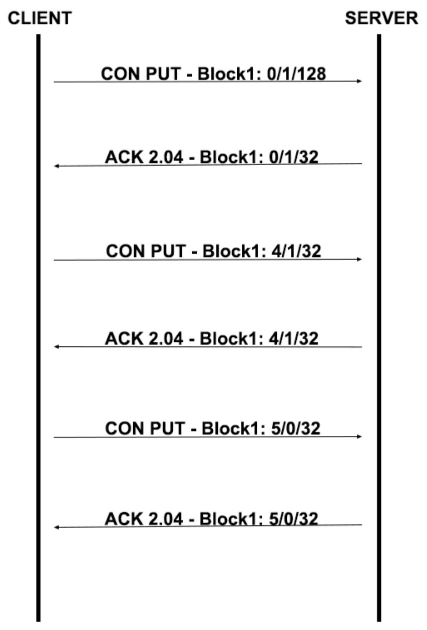
CLIENT
SERVER
CON PUT - Block1: 0/1/128
ACK 2.04 - Block1: 0/1/32
CON PUT - Block1: 4/1/32
ACK 2.04 - Block1: 4/1/32
CON PUT - Block1: 5/0/32
ACK 2.04 - Block1: 5/0/32

Figure 4.1: Block1 example

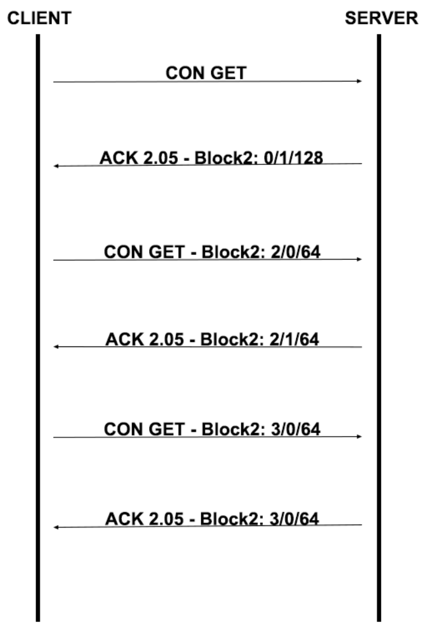
CLIENT
SERVER
CON GET
ACK 2.05 - Block2: 0/1/128
CON GET - Block2: 2/0/64
ACK 2.05 - Block2: 2/1/64
CON GET - Block2: 3/0/64
ACK 2.05 - Block2: 3/0/64

Figure 4.2: Block2 example

4.1.3 Size1 and Size2 options

Other two new options are defined in the draft:

- *Size1*: this option could be used in a request carrying a *Block1* Option, to indicate the current estimate the client has of the total size, or in a 4.13 response (see Section 4.1.1), to indicate the maximum acceptable size;
- *Size2*: this option could be used in a request, to ask the server to provide a size estimate along with the usual response (value 0) or in a response carrying a *Block2* Option, to indicate the current estimate the server has of the total size.

4.1.4 Implementation

Only the client side support for the Blockwise Transfer has been implemented because the need for the implementation of that function emerged during the HTTP-to-CoAP Proxy development, since it acts as a CoAP client, as described in Section 3.6. For the Blockwise Transfer support a *BlockwiseCoapClient* class was created, first used for the proxy and then extracted as a standalone component. This class has a *sendMessage()* blocking method that sends a CoapMessage, waits for the response and returns it to the caller. If a Blockwise Transfer is not necessary then it acts as a normal CoAP client, otherwise it passes the message to one of the two helper classes that handle the Blockwise Transfer: *CoapBlock1Client* and *CoapBlock2Client*. When the message must be sent in chunks is passed to the CoapBlock1Client, that sends all the blocks and return the final response to *BlockwiseCoapClient*. When the response must be received in chunks the first chunk is passed to the *CoapBlock2Client* that collects all the following blocks, builds the complete response and returns it to *BlockwiseCoapClient*. These two helper clients passes the response to the main one through a method called *onBlockTransferComplete()*, method of the *CoapBlockClientListener* interface, so the *BlockwiseCoapClient* implements it and passes a reference of itself to the two Block Clients (Observer design pattern). All that passages are invisible to the user that can only call the public method *sendMessage()* and optionally set the parameters for the transfer: Block1/Block2 preferred size and the Blockwise Threshold.

4.1.5 Using the Blockwise Transfer

The code to start the *BlockwiseCoapClient* with default values is shown in Listing 4.1

```
CoapMessage coapResponse = new BlockwiseCoapClient(coapHost,coapPort,this.coapProvider,
    BlockwiseCoapClient.USE_DEFAULT_THRESHOLD,BlockwiseCoapClient.
    USE_DEFAULT_BLOCK1_SIZE,BlockwiseCoapClient.USE_DEFAULT_BLOCK2_SIZE).sendMessage(
    coapRequest);
```

Listing 4.1: Blockwise client example.

4.2 Resource observing

The basic CoAP messaging model does not work well when a client is interested in having a current representation of a resource over a period of time, so the protocol's implementation can be extended with a mechanism that allows a CoAP client to *observe* a resource on a CoAP server: the client retrieves a representation of the resource and requests this representation to be updated by the server as long as the client is interested. This extension of the protocol is based on the well-known observer design pattern and is described in the observe-draft [9].

4.2.1 Registering and receiving notifications

A client registers its interest in a resource by sending an extended GET request to the server: it is a normal GET with the Observe option, a new option defined in the draft. The value of this new option is an integer, and must be set to 0 for the registration (or a null value for older versions of the draft). The server then stores that request and sends periodic notifications as responses for the received registration message (with the same token), and the observe option with a raising value over time, value that the client can use to reorder the received notifications.

A CoAP server is the authority for determining under what conditions resources change their state and thus when observers are notified: for example a notification could be sent for every change in the resource value or when it reaches a predefined threshold.

4.2.2 Stopping observing resources

The client could send a periodic registration request to update his observer status, and if a notification is CON it must respond with an ACK or it will be removed from the observer's list. When a registration request arrives the client is added to the observers if the response code is 2.xx, and removed from them if it was already registered for that resource and the response code is 4.xx. It is also possible for the client to directly remove his registration by sending a GET request to the resource with the Observe option set to 1.

4.2.3 Implementation

For the Java based implementation of this feature on the server side, the class *ObserveHandler*, which handles all the features described above, has been created.

The handler needs a *ResourceStore* as parameter: it is a store where the resources are saved with the related values. The *ObserveHandler* exposes methods to register and unregister observers for a resource, and there's a *ObserveUtil* class that contains static methods to check the presence and value of the Observe option in a request. The most important method that can be called is the *onUpdate()* method of the *CoapResourceListener* interface: it requires a resource URI as parameter and its call triggers the sending of a notification to all the observers registered for that resource, retrieving the value from the store. The *ObserveHandler* implements the *CoapResourceListener* interface so it is the sender of the notifications. It needs a *CoapProvider* to perform message sending: it must be set with a setter method. An interface has been used so that new handlers for the observing functions could be created without changing the basic functioning of the *onUpdate()* method.

A client willing to register for observing a resource must send a GET request with the Observe option set to 0 to the server and a listener for the token of that request must be set on the client's *CoapProvider*. The client must also implement the *CoapProviderListener* interface so that he can receive notifications, which are handled in the *onReceivedMessage()* method.

4.2.4 Using the Observe option

The code to set up all the classes on the server side is shown in the Listing 4.2, where also a *GetHandler* is implemented: it builds the response to a GET request automatically, by getting the resource value from the *ResourceStore*. The code to handle observers registration on the server side is shown in Listing 4.3. The code to send notifications is shown in Listing 4.4. A last code example shows how to send a registration GET on the client side (Listing 4.5).

```java
this.store = new BasicCoAPResourceStore();
this.getHandler = new BasicCoAPGETHandler(this.store);
IObserverMap observerMap = new ObserverMap();
this.observeHandler = new CoAPObserveHandler(this.store, observerMap);
this.observeHandler.setCoapProvider(this.coapProvider);
```

Listing 4.2: ObserveHandler initialization example.

```java
if(msg.getCode() == CoapMethodCode.GET){
            CoapMessage response = this.getHandler.handleGet(msg);
            if(this.observeHandler != null)
                if(ObserveUtil.isObserveOptionPresent(msg, ObserveUtil.
                    OBSERVE_REGISTER_VALUE) |ObserveUtil.isObserveNullOptionPresent(msg)
                    ){
                    if(response.getCodeAsString().substring(0, 1).equals("2")){
                        //Registering client as observer for that resource
                        if(!this.observeHandler.isObserver(msg.getRemoteSoAddress(),
                            this.servicesListResourceURI)){
                            this.observeHandler.addObserver(msg.getRemoteSoAddress(),
                                BinTools.bytesToHexString(msg.getToken()), this.
                                servicesListResourceURI);
                            response.addOption(new CoAPObserveOption(CoAPObserveHandler.
                                sequenceNumberToBytes()));
                            CoAPObserveHandler.incSequenceNumber();
                        } else {
                            this.observeHandler.removeObserver(msg.getRemoteSoAddress(),
                                 this.servicesListResourceURI);
                            this.observeHandler.addObserver(msg.getRemoteSoAddress(),
                                BinTools.bytesToHexString(msg.getToken()), this.
                                servicesListResourceURI);
                            response.addOption(new CoAPObserveOption(CoAPObserveHandler.
                                sequenceNumberToBytes()));
                            CoAPObserveHandler.incSequenceNumber();
                        }
                    } else if(response.getCodeAsString().substring(0, 1).equals("4")
                            || response.getCodeAsString().substring(0, 1).equals("5")){
                            //Error, unregistering the client if it is an observer
                        if(this.observeHandler.isObserver(msg.getRemoteSoAddress(), this
                            .servicesListResourceURI)){
```

```
                            this.observeHandler.removeObserver(msg.getRemoteSoAddress(), this.servicesListResourceURI);
                        }
                    }
                }else if(ObserveUtil.isObserveOptionPresent(msg, ObserveUtil.OBSERVE_UNREGISTER_VALUE)){
                    //Explicit deregistration
                    this.observeHandler.removeObserver(msg.getRemoteSoAddress(), this.servicesListResourceURI);
                }
            new CoapTransactionServer(this.coapProvider, msg, this).respond(response);
        }
```

Listing 4.3: Observers registration example.

```
this.store.put(resourceUri, new BasicCoAPResource(CoapFormat.APPLICATION_JSON, newJsonValue));
this.observeHandler.onUpdate(resourceUri);
```

Listing 4.4: Notification example.

```
CoapMessage request = CoapMessageFactory.createRequest(false, CoapMethodCode.GET,null);
request.addOption(new CoapOption(CoapOptionNumber.UriHost, serverIp.getBytes());
request.addOption(new CoapOption(CoapOptionNumber.UriPort, CoapUtil.intToBytes(serverPort)));
request.addOption(new CoAPObserveOption(new byte[]{0})); //Should be defined as a constant
new CoapTransactionClient(this.coapProvider, this.serverSocketAddress, this).request(request);
this.coapProvider.addListener(CoapMessageSelector.messageTokenSelector(request.getToken()), this);
```

Listing 4.5: Client side registration GET example.

4.3 Other extensions

During the development of the HTTP-to-CoAP Proxy and of the above mentioned CoAP extensions, other classes and interfaces have been developed, to be added to the mjCoAP library, in order to simplify the extraction of some information from CoAP messages.

4.3.1 CoapFormat

This package was developed to help the programmer in the sending and receiving of the media types as values for the Content-Format and Accept options. The value of these options must be an integer that identifies the type (see Table 4.1).

Media type	Integer ID
text/plain;charset=utf-8	0
application/link-format	40
application/xml	41
application/octet-stream	42
application/exi	47
application/json	50

Table 4.1: Media type identifiers.

With this library it is possible to simply obtain the number as an integer constant from the *CoapFormat* class, or to obtain his byte value from the enum *CoapFormat-Type*. Is it also possible to add the Accept and Content-format options to a message using the *CoapAcceptOption* and *CoapContentFormatOption* classes, or with the normal *CoapOption* class. The first implementation uses the extended options and the enum type, and is shown in Listing 4.6. The second uses the basic *CoapOption* and the *CoapFormat* constants and is shown in Listing 4.7.

```java
//Adding the content format option:
coapMessage.addOption(new CoapContentFormatOption(CoapFormatType.TEXT_PLAIN));
//Extracting the option:
o = new CoapContentFormatOption(CoapUtil.getCoapOption(coapMessage,CoapOptionNumber.
    ContentFormat).getValue());
if (o.equals(CoapFormatType.TEXT_PLAIN))
    System.out.println("Text plain format!");
```

Listing 4.6: CoapFormat example, using the enum CoapFormatType.

```java
//Adding the content format option:
coapMessage.addOption(new CoapOption(CoapOptionNumber.ContentFormat, new byte[]{
    CoapFormat.TEXT_PLAIN_UTF_8});
//Extracting the option:
System.out.println("Content-format: " + CoapFormat.toString(CoapFormat.fromBytes(
    CoapUtil.getCoapOption(coapMessage, CoapOptionNumber.ContentFormat).getValue())));
```

Listing 4.7: CoapFormat example, using the class CoapFormat.

4.3.2 CoapUtil

CoapUtil is a package grouping a set of of static classes created to cope with some mjCoAP library limitations. These problems mainly derive from the fact that option values are represented as byte arrays, which make it possible to violate the format intended for a given option. For instance, the *Uri-Port* option expects an integer value, but the library allows to insert a string value as a byte array. In order to solve these issues, the following methods have been implemented:

- *int bytesToInt(byte[] value)*: converts a byte array (of any length) to an Integer number. For instance, if the byte array contains 22,51 the number will be 22*256 + 51 = 5683.

- *byte[] intToBytes(int value)*: converts an integer number into a byte array. For instance, if the number is 5683 the byte array will contain 5683 div 256, 5683 mod 256 (div is the integer division).

- *CoapOption getCoapOption(CoapMessage m, int optionNumber)*: extracts a CoAP option from the message m, with the selected option number. In the library there is only a method that extracts the String value of a CoAP option from a message, or another one that extracts an array with all the options, but there isn't a method to extract a single CoAP option returned as a *CoapOption* object. It is necessary because if an Integer value is needed it's not possible to extract it from the String value.

- *LinkedHashSet<CoapOption> getCoapOptions(CoapMessage m,int optionNumber)*: like the method above, but extracts a List of options with the selected option number. The library doesn't contain a method to extract a multiple CoAP option but it is necessary: for example to retrieve all the Uri-Path or Uri-Query options and rebuild the request URI (see Section 2.2).

- *String getMessagePath(CoapMessage msg)*: this method extracts all the Uri-Path options and returns the whole request path as a String.

- *String getMessageQuery(CoapMessage msg)*: this method extracts all the Uri-Query options and returns the whole request query as a String.

Chapter 5

Experimental evaluation

The first phase of the evaluation and testing process was concentrated on separated tests for the single classes and libraries:

- **HTTP-to-CoAP Cross-Proxy**: the proxy was tested by creating a Java CoAP test server simulation of a parking lot scenario. In that scenario a CoAP server contains a list of parking spots that can be *free* or *busy* and handles the single parking's status modification via POST message. With a GET at */parkings* the whole list is returned, with the status of every parking, and with a GET at */parkings/N* the response is the status of the parking spot number N.

  For the test, the described Parking Lot Server and the HTTP-to-CoAP Proxy were started, and then GET requests were performed to the server via HTTP, through the Proxy, using DHC as HTTP client (see Section 2.4.1). The parking's status could be modified using Copper (see Section 2.3.3) or a Java CoAP client;

- **Blockwise Coap Client**: For the testing of the BlockwiseCoapClient it was used to perform requests to a Contiki node (see Section 2.3.4), set up by another student, that responded with a message split in chunks. A test server was programmed in Java to verify all the possible Blockwise Transfer cases;

- **Observe:** To test the observe package a Java server that implemented the ObserveHandler was created, as described in Section 4.2, and then his resources were observed using different clients like Copper or a Java based one;
- **CoapFormat and CoapUtil**: these two packages are linked and used as external libraries in all the other projects so their test was simply programming and testing the CoAP extensions listed above and described in all the previous chapters.

5.1 Demonstrative IoT Scenario

The final test of the components was to build a complex architecture to demonstrate the power of the Internet of Things and the interoperability among elements using standard protocols. The development of all the software for the components has been done by a team of B.Sc., M.Sc., and Ph.D. students, at the WASN Lab (Wireless Ad hoc Sensor Network Laboratory) of the Department of Information Engineering of the University of Parma.

This demo was then presented as a project of the WASN Lab at the SPS IPC Drives Italia 2014[1] in Parma, May 20-22, 2014, and at the European Conference on Networks and Communications (EuCNC 2014) in Bologna, June 25-27, 2014. An illustrative representation of the architecture is shown Figure 5.1, followed by a description of the various components.

[1]http://www.spsitalia.it/

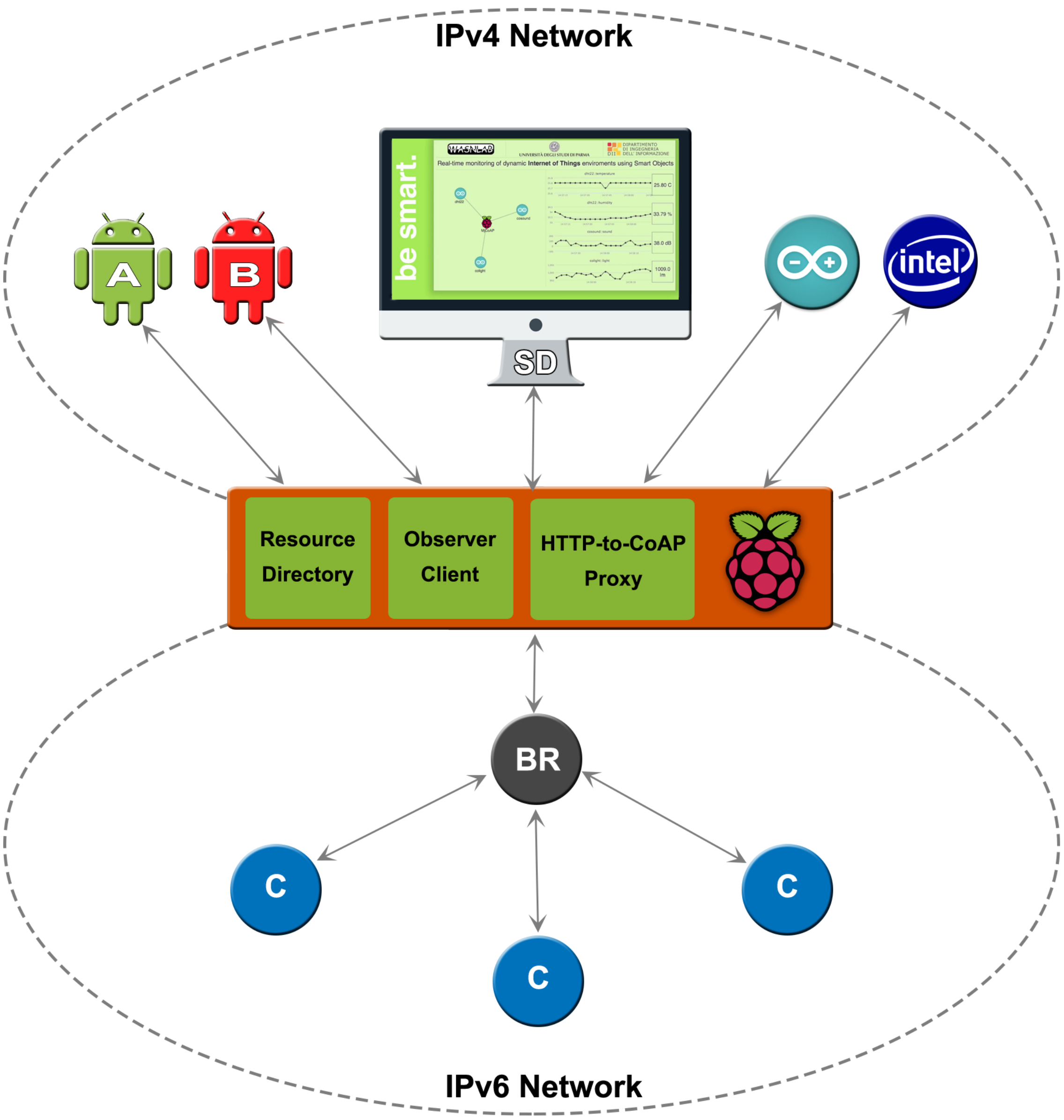

Figure 5.1: Demonstrative IoT architecture created at the WASN LAB.

The basic idea is that a sensor in the architecture, when started, sends a POST message to the *ResourceDirectory* (RD) with a list of the services that it offers, and the RD must send the services list to the *ObserverClient* (OBS) every time a new service is added: the OBS is registered to the RD's services list resource.

The OBS then registers to all the services on the list and their values (received as notifications) are posted real-time on a *Smart Display*. The detailed descriptions of the components are described next.

- **Raspberry Pi**: represented in the image by the orange box in the middle, it's a single-board computer developed in the UK from the Raspberry Pi Foundation. In this case is used to run the Resource Directory, the Observer Client and the HTTP-to-CoAP Proxy (as .jar files). It's connected to an IPv4 network via Ethernet (direct cable to the network's router) and to the IPv6 IEEE 802.15.4 network's border router, where the IPv6 network contains a series of nodes running Contiki (see Section 2.3.4).

- **Resource Directory (RD)**: it is a Java CoAP server that offers as a resource the list of the services, published with a specific path (e.g. */services*). If it receives a POST message with a JSON list of URIs, addressed to the list's path, it adds them to the services list with also their http URI (Proxy's base URI and concatenated CoAP URI). If it receives a GET addressed to the list's path it returns the whole list, that is also observable. The handling of observers and notifications is based on the library developed for this thesis.

- **Observer Client (OBS)**: it is a Java CoAP client that performs a registration to the services list of the RD. When an updated version of that list arrives (in a notification) the OBS sends a registration GET to all the new listed services, and after that it will receive periodic values from all these resources, via notification, and it will post them to the Smart Display. In practice, every time that a new service is present in the list the node related to its IP is added to the display, if not present, and then the service is attached to that node waiting for the periodic values that will be displayed on the graph.

- **HTTP-to-CoAP Proxy (HC)**: it is the proxy developed for this thesis and can be used to send CoAP requests to the nodes via HTTP. In this scenario it is also the only way to perform requests from the IPv4 network to the IPv6 one.
- **Smart display (SD)**: the smart display, developed by the WASN Lab, is a web application that handles http POSTs by adding nodes and graphs of the related services. The messages are handled by a Smart Display Proxy(a Java application).
- **Contiki border router (BR)**: it is the border router for the IPv6 network of Contiki nodes.
- **Contiki nodes (C)**: nodes with Contiki OS, every node offers some services and periodically posts them to the RD. It also responds to GET requests that arrives from the OBS (with registration) or from the HC, and sends periodic notification to observers (like the OBS once registered) with the value of the resources. The services offered by these nodes in the demo were temperature, light and movement, and their values were retrieved through sensors attached to them.
- **Arduino Yun (∞)**: it has the same behaviour of the Contiki nodes, but on the IPv4 side of the architecture, and it's programmed in Python. In the demo it offered a temperature sensing service.
- **Intel Galileo**: it has the same behaviour of the Arduino Yun, but programmed in Java. In the demo it offered a temperature sensing service.
- **Android app A**: it is a CoAP server with the same behaviour of the Contiki,

Arduino and Galileo nodes, but implemented as an Android application. In the demo it offered a noise sensing service.

- **Android app B**: it is an Android application that shows a list of the services on the network, obtained via GET request to the resource directory, and can perform a GET request to their HTTP version through the HTTP-to-CoAP Proxy, or to their CoAP version only if the node is reachable on the IPv4 network.

Chapter 6

Conclusions and future work

The objective of this thesis was to create an HTTP-to-CoAP Proxy for the interoperability among application-layer protocols in IoT scenarios. The Proxy, has been developed as a Java application based on the mjCoAP library, it performs protocol translation, and it respects the main guidelines of the related draft. The http-mapping-draft is still incomplete because it is at an early version (3 at the moment), and, therefore, a future work will be the proposal of some changes based on the experience and the implementation choices made during the development of the Proxy.

The development includes support for Blockwise Transfer (an option mentioned in the draft). Because of that, and also because this kind of transfer is not supported by the mjCoAP library, a client with that feature has been implemented. It was developed following the related draft and implemented as a standalone component to allow its use as a CoAP client also outside the proxy. A further thing to do could be to implement the support for the server side handling of the Blockwise Transfer, and also to integrate both the sides implementations in the mjCoAP library, possibly in a transparent way for developers. For instance, these functions could be integrated in the methods *request()* and *onTransactionResponse()* and the programmer must only choose if activating or not the Blockwise Transfer, and setting its parameters.

After the implementation of the HTTP-to-CoAP Proxy and Blockwise CoAP client, support for the CoAP Observe option has been implemented. This library was developed following the related draft and it became very useful in the laboratory, since it was used as base of the demonstrative scenario described in Chapter 5. Some further work related to that library could be its integration in mjCoAP and also the support of observing function with Blockwise Transfer, described as a possible implementation in the Blockwise Transfer draft.

Moreover, the CoapUtil and CoapFormat libraries have been implemented in order to cope with some limitations of the mjCoAP library. These libraries should be integrated in the next version of mjCoAP.

In conclusion, all the software modules implemented in this work have been tested and played a central role in the demonstrations made by the WASN Lab at the SPS IPC Drives Italia fair and the EuCNC 2014 Conference. The demonstrative testbed also involved components developed by other students and researchers of the laboratory, and proved the real power of the Internet of Things based on the use of standard protocols (CoAP, IP, RPL, IEEE 802.11, IEEE 802.15.4) with the integration of heterogeneous devices, in terms of hardware platforms, connectivity, operating systems, and implemented functionalities.

Bibliography

[1] Cisco, Lopez Research LLC, *An Introduction to the Internet of Things (IoT)*, http://www.cisco.com/web/solutions/trends/iot/introduction_to_IoT_november.pdf, November 2013.

[2] Wikipedia, *6LoWPAN*, http://en.wikipedia.org/wiki/6LoWPAN#cite_note-3.

[3] Standard track, *rfc6550*, http://tools.ietf.org/html/rfc6550#page-154.

[4] Internet rfc - core working group, *rfc7252*, http://tools.ietf.org/html/rfc7252.

[5] Cisco, Lopez Research LLC, *Building Smarter Manufacturing With The Internet of Things (IoT)*, http://www.cisco.com/web/solutions/trends/iot/iot_in_manufacturing_january.pdf, January 2014.

[6] Cisco, Lopez Research LLC, *Smart Cities Are Built On The Internet Of Things*, http://www.cisco.com/web/solutions/trends/iot/docs/smart_cities_are_built_on_iot_lopez_research.pdf.

[7] Roy Fielding *Representational State Transfer*, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[8] Internet rfc, *rfc3986*, `http://tools.ietf.org/html/rfc3986`.

[9] Internet draft - core working group, *draft-ietf-core-observe-14*, `http://tools.ietf.org/html/draft-ietf-core-observe-13`.

[10] Internet draft - core working group, *draft-ietf-core-block-14*, `http://tools.ietf.org/html/draft-ietf-core-block-14`.

[11] Internet draft - core working group, *draft-ietf-core-http-mapping-03*, `http://tools.ietf.org/html/draft-ietf-core-http-mapping-03`.

[12] Wikipedia, *Contiki*, `http://en.wikipedia.org/wiki/Contiki`.