Aalto University
School of Science
Degree Programme in ICT Innovation

Anastasiia Karpenko

Practical Use of O-MI/O-DF messaging standards in mobile application for IoT.

Creating an open system for smart EV charging.

Master's Thesis
Espoo, September 18, 2017

Supervisors: Professor Kary Främling, Aalto University,
Professor Maurizio Marchese, University of Trento
Advisor: M.Sc. Andrea Buda

Aalto University
School of Science
Degree Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author: Anastasiia Karpenko

Title:
Practical Use of O-MI/O-DF messaging standards in mobile application for IoT. Creating an open system for smart EV charging.

Date: September 18, 2017 **Pages:** 78

Major: Service Design and Engineering **Code:** SCI3022

Supervisors: Professor Kary Främling, Professor Maurizio Marchese

Advisor: M.Sc. Andrea Buda

During the last decades the Internet has become ubiquitously available in most places of the world, which has made it possible also to implement the Internet of Things (IoT) paradigm. In this paradigm Internet connects devices with each other and with their users allowing the development of digital services and applications that increase the comfort level of everyday human life. Many domains are interested to exploit the IoT ecosystem, especially public administrations starting *Smart City* initiatives all over the world. Cities are becoming smart in many way: smart mobility, smart buildings, smart environment and so on. However, the problem of noninteroperability in IoT exists that hinders the seamless communication between all kinds of IoT devices. Different domain specific IoT applications use different messaging standards offered by different providers. These messaging standards do not comply with each other. The Open Group published two domain-independent standards O-MI and O-DF aiming to solve this interoperability problem. In this thesis we want to describe the practical use of O-MI/O-DF standards in a mobile application for the smart city context, in particular for the Smart Mobility domain, electric vehicle (EV) charging use case.

First, the overview of IoT domain and its ecosystem with attention to noninteroperability problem is made. Then the description of six messaging standards including Open Group's standards O-MI and O-DF are provided. Then the requirements for IoT messaging protocol are outlined and the comparison of all these messaging standards are made showing that the domain independent standards O-MI/O-DF are the most suitable for IoT. After that smart city context and its requirements are described for the implementation part of the thesis. In the next chapter the implementation of a mobile application using O-MI/O-DF messaging standards are described with the details of the application architecture, structure of messages and overview of the back-end part of the service.

Keywords: Internet of Things, O-MI, O-DF, Smart Cities, Smart Mobility, EV charging, Messaging Standards, BIoTope project

Language: English

Acknowledgements

This master thesis was written for the Master of Science degree in ICT Innovation with major in Service Design and Engineering.

My personal motivation for this thesis project was to learn how to develop mobile applications for Android OS and to know more about IoT domain. I always was interested in Smart City concept and I am very happy that this thesis was a part of bigger project BIoTope that is directly related to Smart City domain.

I want to express my gratitude to my supervisor professor Kary Främling for a chance to join this project, for guidance and support. It is an honour to do the thesis project in the worldwide IoT pioneer group. I would like to thank my colleagues from ASIA research team for advises, help and support in this project, especially Andrea Buda, Antti Nurminen, Bhargav Dave, Asad Javed, Narges Yousefnezhad, Manik Madhikermi. I was glad to collaborate with the other programmers of BIoTope project: Lauri Isojärvi, Tuomas Kinnunen and Jussi Pirilä. I want to thank them all.

At last I would like to express my gratitude to my mother and brother for their love, motivation, encouragement and support.

Espoo, September 18, 2017

Anastasiia Karpenko

Abbreviations and Acronyms

AMQP	Advanced Message Queueing Protocol
API	Application Program Interface
ARPANET	Advanced Research Projects Agency Network
bIoTope	Building an IoT Open Innovation Ecosystem for Connected Smart Objects
CoAP	Constrained Application Protocol
CRUD	Create, read, update and delete
CSV	Comma-Separated Values
D2D	Device-to-Device
D2S	Device-to-Server
DDS	Data Distribution Service
DOM	Document Object Model
DTSL	Dynamic Technologies Services Limited
EU	European Union
EV	Electric Vehicle
GPS	Global Positioning System
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ICT	Information and Communications Technology
IETF	Internet Engineering Task Force
IoT	Internet of Things
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
MobiVoc	Open Mobility Vocabulary
MQTT	Message Queue Telemetry Transport
NAT	Network Address Translation
O-DF	OpenDocument Format
O-MI	Open Messaging Interface
OMG	Object Management Group
POJO	Plain Old Java Object

QoS	Quality of Service
RDF	Resource Description Framework
REST	Representational State Transfer
S2S	Server-to-server
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol
SoS	Systems-of-Systems
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	Time-to-Live
UDP	User Datagram Protocol
URL	Uniform Resource Locator
US	United States
USB	Universal Serial Bus
W3C	The World Wide Web Consortium
Wi-Fi	Wireless Fidelity
WWW	World Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Contents

Chapter 1

Introduction

This Introduction describes the background and theoretical overview about the research topic. Then we introduce the research objectives and research framework, followed by an overview of the structure of the thesis.

1.1 Motivation

The first network ARPANET was developed in 1969 [7] and it was the beginning of Internet. First it was developed for military [2] industry and contained only 2 members but it grew to 213 members during the next 12 years [13, 15] attracting members mainly from the US universities. In the 90s personal computers became used by more and more people along with active adoption of TCP/IP (Transmission Control Protocol/Internet Protocol), HTTP (Hypertext Transfer Protocol) and HTML (Hypertext Markup Language). All of it lead to establishment of World Wide Web (WWW).

For the last 20 years, Internet became widely used in the world [45]. Internet became the center of the communication paradigm called the Internet of Things (IoT). In this paradigm, the everyday life objects communicate with each other and with their users via Internet, they have micro controllers, transceivers and messaging protocols that allow their communication[1].

IoT emerged from domains, such as logistics, where it was necessary to organize the product tracking. IoT was first mentioned in 2002, in the scientific article by Huvio, Grönval and Främling [17]. They created a lightweight distributed system to share information by using peer-to-peer connections for parcels tracking. A year later, as a variant of IoT implementation, Främling, Holmström, Ala-Risku and Kärkkainen proposed to use the agent-centric solution, in which each product should have a valid link to its data, available in Internet during the whole product's life cycle. The agent handles infor-

mation requests and maintenance associated with each physical product in the supply chain [9].

IoT allows the development of a huge variety of applications that will use the enormous quantity of data generated by IoT devices such as cameras, home appliances, sensors, actuators, vehicles and so on. These applications can allow citizens, companies and public administrations to benefit in everyday life by using new digital services that are built on top of IoT ecosystem. Domains such as home automation, industrial automation, medicine, energy management, smart grids, automotive, traffic management and many others will depend on IoT services and applications [3].

Nowadays, many national governments are willing to adopt digital solutions in the management of public affairs trying to make the concept of *Smart City* a reality [35]. So urban context is of a particular interest where IoT can be applied.The concept of *Smart City* implies that the digitization of services can increase the quality of life of the citizens and also reduce the operational costs of public administrations.

As a result, there will be a stimulus to create new services using IoT technologies and ecosystem, where the amount of useful data created by these services can be used to increase transparency and show the actions of local governments to the citizens. Specifically, IoT can bring benefits to public services as lighting, surveillance of public areas, garbage collection, hospital services, educational services as well as transport and parking services [8].

The Smart City market is estimated to reach hundreds of billions of dollars by 2020 with annual spending around 16 billions [31]. This market is formed by the interconnection of industry and service sectors that are important for citizens, such as Smart Governance, Smart Mobility, Smart Buildings, Smart Utilities and Smart Environment. However, Smart City market has not really taken off yet because of a number of different reasons: political, technical and financial. The most important technical reason is noninteroperability of various technologies used in the city and urban development. In this case, IoT vision can help to implement an integrated ICT platform for cities and urban environments and release the full potential of Smart City concept [16, 28].

1.2 Research Objectives and Framework

Interoperability remains one of the main issues for IoT. There is a need for standartization of the whole field to make it possible to create an open IoT system for Smart City. Such a system will provide an open platform where a big number of services offered by different service providers can co-

exist, collaborate and communicate with each other. A particular problem exists with the messaging protocols for IoT. There is a number of messaging standards that have been used by different IoT - based service providers. These standards have different characteristics and each of them provides the best possible quality of service for particular applications in different industry fields. However, they can hardly be interoperable across these fields.

A number of criteria for IoT messaging standard has been specified [10, 11, 24]. Some of the existing standards (like the domain-independent IoT messaging protocols O-MI/O-DF, specified by the Open Group), are generally more suitable for IoT according to the existing requirements.

The aim of the current thesis is to apply the most suitable IoT messaging protocols to the Smart City domain and to implement a mobile application based on the potentially opened IoT system that can be used by the administration of any city. One of the ongoing smart city projects in Europe, bIoTope, provided the context for this implementation. This project's goal is to build an IoT open innovation ecosystem for connected smart objects [40]. Its smart mobility use case had a need for a proof - of - concept. The bIoTope project offered an opportunity to make the implementation of a mobile application in the context of smart parking and Electric Vehicles (EV) charging field.

According to the background and framework that we have just introduced we defined the following **research questions**:

1. Which messaging standards for IoT exist?

2. What are O-MI and O-DF standards, how are they different from the other existing messaging standards and how well do they suit all the requirements for messaging standard specific to IoT?

3. How can O-MI/O-DF standards be implemented within a mobile application that can be used in a Smart City context, especially in the smart mobility (parking and EV charging) use case?

The research questions were formed formed due to the necessity to have a general standard for messaging in IoT applications that will satisfy all the specific requirements of IoT.

The thesis has three main **objectives**:

1. To describe O-MI/O-DF messaging standards and to show how they can enable interoperable communication between devices in IoT.

2. To develop a mobile user interface (mobile application) for Android OS that will call O-MI node and use O-DF format to send to and get information from it. This mobile application will represent an open IoT system that can accept publishers and consumers of services that use O-MI/O-DF standards.

3. To test the mobile application in the context of the bIoTope project, specifically its smart EV charging use case.

There are several main **guidelines** to proceed the work with these objectives:

- Background topic discussion that will be focused on the discussion about what is Internet of Things and smart objects, IoT ecosystem, mostly used messaging protocols for IoT.

- Context introduction: overview of bIoTope project and particular EV charging use case.

- EV Charging mobile application design and implementation.

1.3 Structure of the Thesis

The thesis consists of eight chapters. The chapter that follows the introduction provides an overview of Internet of Things and smart objects. It shortly describes the ecosystem of the IoT and points out the main issue in the IoT area, interoperability issue. Chapter 3 describes the mostly used messaging protocols in IoT. Chapter 4 describes O-MI/O-DF - the messaging standards designed by The Open Group. In chapter 5, we present the requirements for IoT messaging and comparison framework within which the existing IoT messaging protocols will be compared in order to figure out which of them is the most suitable one. Chapter 6 introduces the bIoTope project and the case study which will relate to the implementation part of the current thesis. Implementation of the software system (in particular, a mobile application for Android OS) is described in the chapter 7. Chapter 8 concludes the thesis.

Chapter 2

Internet of Things and Smart Objects

The Internet of Things (IoT) is a fast growing wave of the Internet development. While in the 90's 1 billion of people were connected over the fixed Internet and 10 years later another 1 billion of people got connected over mobile Internet, by the year 2020 there is a potential to connect 28 billion things and by the year 2050 already 50 billion things (such as bracelets, cars or home utensils) to the Internet by the means of IoT technologies [18, 25]. The low cost of sensors, good processing power and bandwidth to connect things create ubiquitous connections. IoT becoming more used in the day-to-day life with the simple products like fitness trackers and smart thermostats gaining popularity among the users [18].

The paradigm of IoT [12] evolved from the concept of ubiquitous computing [47] to the model of the object system where the objects are loosely coupled and decentralized. Each of them has features of sensors, processors and networking capabilities [1, 22]. IoT uses different means of communications: wired or wireless network connection. More and more physical entities that belong to the system and are familiar to the users in everyday life are becoming smart.

2.1 Introduction to IoT and Smart Objects

The smart objects are digital equipment or household utensils that people normally use in their life routines. "Smart" in this context means that the object can make its own decisions and change behaviour according to the data that it receives from the surrounding environment or the transferred data from other objects sensors that are connected over the network, or data

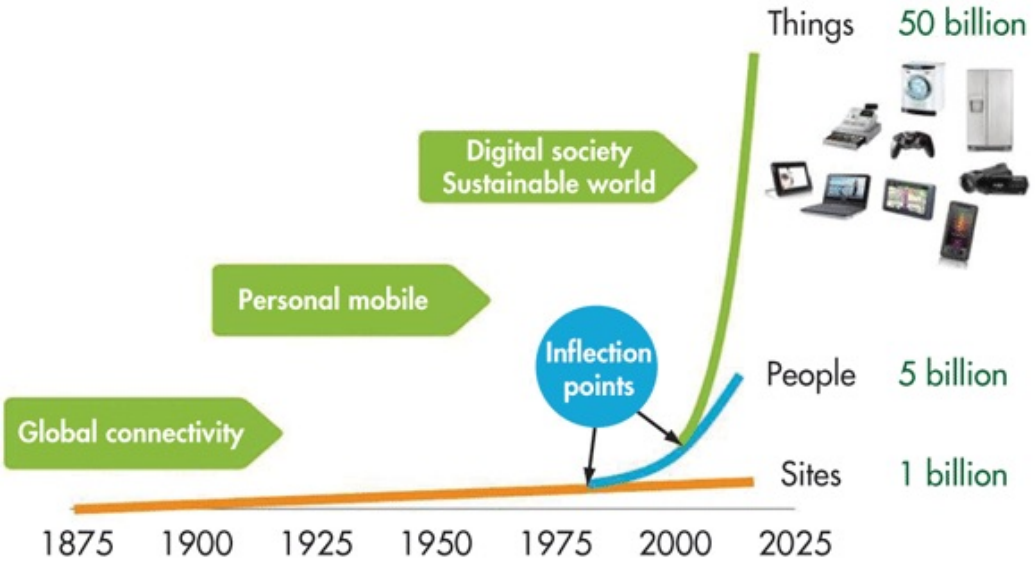

Figure 2.1: Evolution of connected things [25].

that the object received as the outcome of the interactions with the end-users.

IoT environment consists of many networked devices: sensors, data processors, actuators, personal mobile devices, consumer electronics, multimodal systems. Personal mobile devices such as smart phones, tablets and other wearables form the essential class of smart objects. Smart objects function in their micro-environment and organize data flows originated from multiple sources and are consumed by many applications. Smart objects provide the hardware layer of "smart services" - service-oriented applications. These applications are built within the system of objects (agents) that interact with each other making the IoT environment "smart".

Smart environment can get the knowledge about the environment and the users and apply it for the sake of service improvement and to make the quality of user experiences in this environment better [22]. The rich amount of potentially available information nurtures the IoT environment and provides this degree of smartness to the services that are built on top of it. The smart objects can adapt to the changing physical conditions of their environment, can read the state of the environment and see what is happening to it as the result of user interactions with the environment.

The smart object operates in a well-defined cycle: 1) understanding the current state of the environment, 2) applying application goals to the state of the environment and reacting to the results of possible actions of the user, 3) acting upon the environment with the attempt to change its state. A smart

object can provide several smart services. Each service consists of direct observations of the environment and interactions with other smart objects of the system. It can be illustrated by an example of the indoor climate control system that measures the temperature in the room and reacts on the temperature change by decreasing or increasing of the air flow.

A smart service can have more advanced interaction model where information is received and processed by cooperative activity. Several smart objects get and share information to each other about their environment and all the processes that happen inside the environment. The service uses so called multi-agent system where each agent can publish and retrieve information into/from the shared information space[29].

An application is a system of agents distributed over the environment that continuously interact with each other directly by sharing information. Smart objects can host these agents inside of them, like it would do server and desktop computers. If the service is related to several smart objects it means that the service has collective providers where each object can send its own piece of information to commonly shared information space. The object even may not know how and by who this information will be used. All of the participating objects may acquire and use collectively the knowledge about environment and its users. They can supply this information to the multiple services that may apply it to many possible use cases.

As we mentioned earlier these smart objects of IoT are the result of development of ubiquitous computing concept. The multiple technologies of this concept, embedded sensors, communication and protocols make the IoT work but at the same time they raise a lot of challenging questions connected to standardization of communication protocols. In order to understand better which protocols are part of IoT we will consider the short overview of IoT ecosystem.

2.2 IoT Ecosystem

The IoT ecosystem has 7 layers as shown in Figure 2.2. The bottom layer represents the application domain or market where the devices of IoT may be used. It can be smart grid or smart home or digital health business area as an example. The second layer is the layer of smart devices that can be used in the smart space (various sensors, cameras, GPS, smart meters etc). Humidity and temperature sensors can be used to show the climate inside the smart home and video cameras and smart locks can provide security to the house owners.

The third layer is interconnection layer that is responsible for sending the

data from the smart devices (sensors) to the computing facility or a cloud. On the next layer the data is accumulated, sorted and combined according to the themes (data from sensors, population data, traffic data etc). On the next higher "analytics" layer this data is analyzed using various techniques like machine learning and data mining. The next layer is application layer that ensures that large distributed applications will have suitable collaboration and communication software such as software defined networking (SDN) and service oriented architecture (SOA) [34].

The top layers in the ecosystem represents all the services provided on the market: health management, energy management, smart transportation, digital education etc.

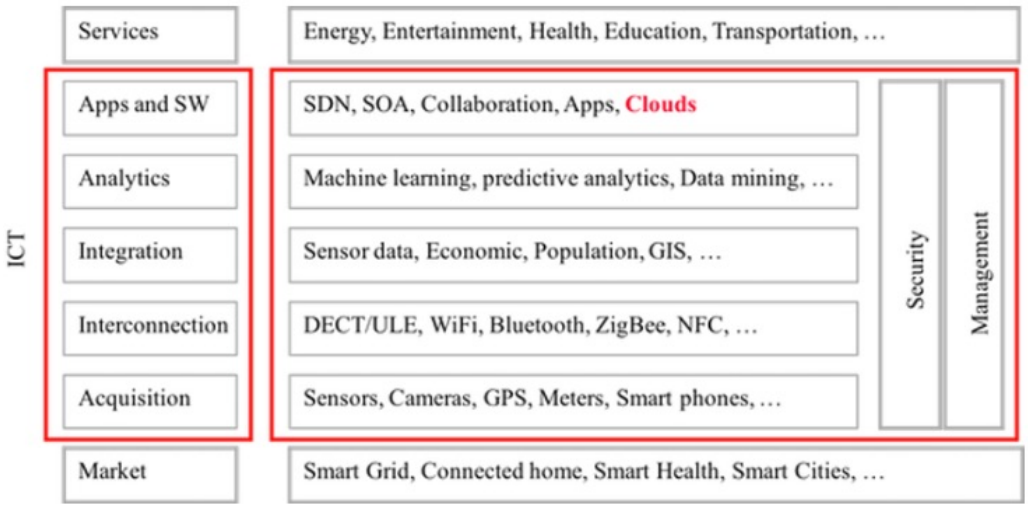

Figure 2.2: IoT Ecosystem [34].

In this thesis we will concentrate on interconnection layer. This layer also has layers with different protocols on each of them as Figure 2.3 shows: datalink layer, network layer, session layer, security and management layer. Datalink layer consists of the the protocols that connect two IoT devices to each other or to the gateway to the cloud. Network layers protocols are dedicated for routing among the sensors and communicating and aggregating the information before sending it to the cloud. The session layer consists of protocols that provide messaging among the devices and actors of the IoT communications. There are also a number of security and management protocols [34].

For the multitude of things in IoT there are multitude of the protocols. In the scope of current work we will focus on the session layer with its messaging

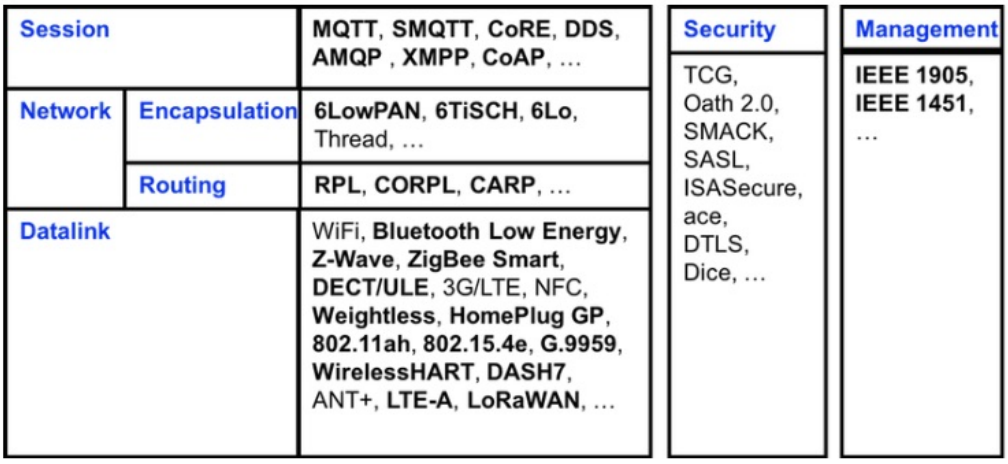

Figure 2.3: Protocols for IoT [37].

protocols for this many IoT smart devices and smart services that have very low communication. Because of it the information that is provided by one service or device is rarely accessible to another service or device. It results in high fragmentation and shows the problem of interoperability between smart devices and services. Interoperability is needed to develop user-centered services that will use the necessary information generated by other participants in the smart environment and present it in personalized and context-aware way to satisfy the target group of users[23].

In the next subchapter we will discuss the interoperability problem and how the messaging standards can be a solution to it.

2.3 Interoperability Issues in IoT

Ideally the IoT ecosystem creates "seamless" programmability of each device or sensor giving the maximum potential of connected experience. That requires interoperability. It means that IoT has to have standards make different platforms to communicate, be operated and programmed across the multitude of devices without any concern of the producers, platforms, operating systems, model, version or industry. The type of device, browser, screen type, hardware should not affect the connectivity between things, people and processes.

But in the reality the big problem with interoperability exists. The reasons of this problem are the following: there are many types of devices man-

ufactured by different producers and cannot be integrated with each other, software cannot be run on the different operation systems, different versions of the software or hardware, communication protocol standards are different across all the things in IoT, different connectors and connectivity frameworks [32].

The above mentioned reasons are also influenced by the technological giants (such as Google, Apple, Microsoft, Samsung, IBM and many more) that contribute to interoperability challenge because they support their individual operating systems, proprietary protocols and standards. These companies are trying to promote and protect their own technologies and solutions that creates the burden to the development and consumerization of IoT [44].

The solution to this problem can be creation of messaging standards for IoT. There is already a set of existing messaging protocols and standards that exist at the session sublayer of interconnection layer in IoT ecosystem that we mentioned before. In the next chapter we will describe the most used of them.

Chapter 3

Messaging Protocols in IoT

With lots of things connected to Internet there are number of messaging protocols that use different new types of communication. They are not limited to the traditional reply-request human-to-device communication but also use device-to-device, device-to-server, server-to-server communications that are going to be widely used in IoT [37].

Device-to-device (D2D) communication is for devices to communicate with each other. Device data then will be collected and send to the server (device-to-server D2S). Then the server will have to share this data with server-to-server (S2S) type of communication providing the data back to devices, other applications and people [37].

The Figure 3.1 describes the basic use cases of protocols on the different level of communication. Each of the protocols are mainly used in the defined patterns of communication. For instance, MQTT protocol is mainly used for collecting device data and sending it to the servers, XMPP protocol connects devices to servers and then to people since people are connected to servers. Both MQTT and XMPP are used for D2S communication pattern. DDS and CoAP protocols represent the device-to-device (D2D) communication pattern, AMQP protocol was designed to connect servers to each other (server-to-server S2S) [37].

These protocols are widely used in IoT and claim to be real-time IoT protocols with publish-subscribe architecture that can connect millions of devices [37]. In this subchapter we will describe a number of messaging protocols that are nowadays widely used in IoT applications.

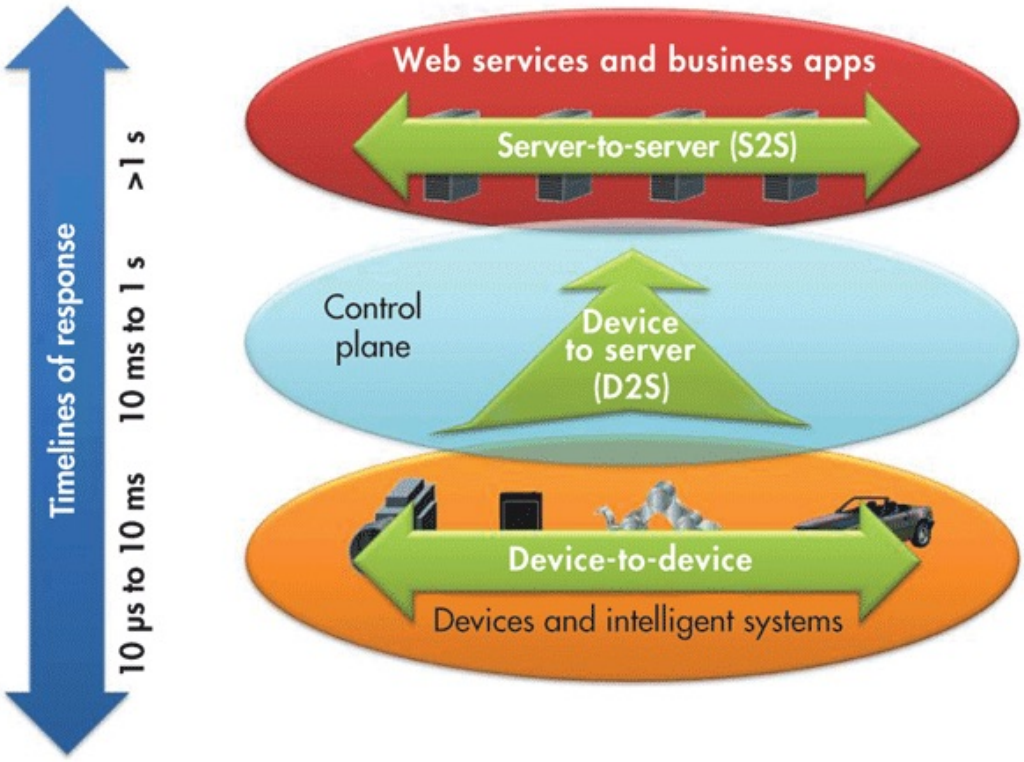

Figure 3.1: IoT protocols need to address response time [37]

3.1 MQTT

Message Queue Telemetry Transport (MQTT) is a protocol designed in 1999 by IBM for usage by constrained devices in low-bandwidth and unreliable networks. It is aimed to collect the data from IoT devices in one place and make it accessible by the enterprise IT infrastructure [41].

The principles of MQTT design aimed to provide more reliability of data delivery in response to device constrained resource requirements and bandwidth. It should provide embedded connectivity between 2 sides: 1) middleware and applications and 2)networks and communications. Architecture of MQTT is organized around publish/subscribe principle (Figure 3.2).

The system contains three component players: publisher, subscriber and a broker. In IoT we can define the publishers as lightweight sensors that send their data to connected broker and go back to the sleeping mode right after the data has been sent. Subscribers are applications that are interested to consume some certain sensory data and they subscribe to the broker in order

to be informed when the new data is available. The brokers sort and classify the data according to the topics and send the data to subscribers who are subscribed to that topic accordingly [34].

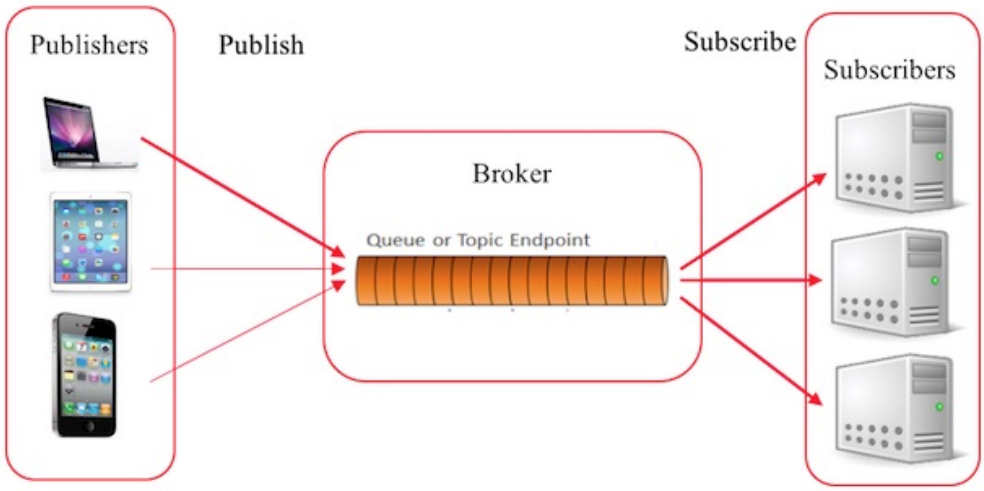

Figure 3.2: MQTT Architecture [34].

MQTT is quite lightweight however it can have several shortages. MQTT requires TCP/IP support that can give additional complexity and difficulty while using small and computationally weak devices (for example, building sensor networks). Another problem can be the message broker. Since it is in the middle of MQTT architecture, it can become a point of failure of the system if something is wrong with the broker.

3.2 CoAP

CoAP stands for Constrained Application Protocol. The purpose of this protocol is to provide lightweight RESTful (HTTP) interface for IoT. It was designed by IETF Constrained RESTful Environment (Core) working group. HTTP client and servers usually use RESTful interface (Representational State Transfer) as a standard interface for communication. REST can result in significant overhead and power consumption for lightweight IoT applications. CoAP can enable low-power IoT devices such as sensors operate and use RESTful services within their power constraints [34].

CoAp is build over User Datagram Protocol (UDP) that is perfect for time-sensitive applications and real-time systems instead of TCP that is com-

monly used in HTTP [25]. CoAP has light mechanism and at the same time provides reliability for device-to-device communication for devices with limited amount of memory available [38].

CoAP architecture has two layers: messaging layers that is responsible for reliability and creation of messages and request/response layer that provides communication. CoAP has different messaging types: confirmable (represents reliable transmissions), non-confirmable (not reliable transmissions), piggyback and separate types that are used request/response communication (Figure 3.3).

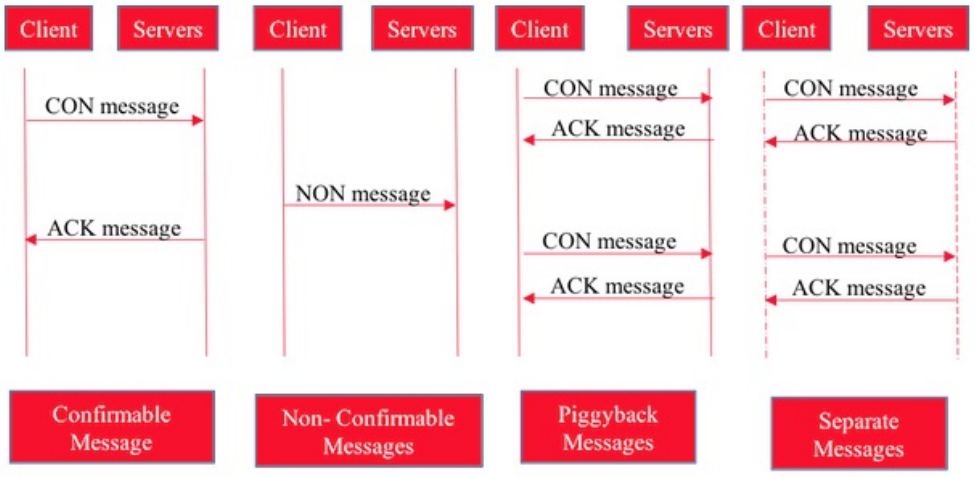

Figure 3.3: CoAP messages [34]

Piggyback type of messaging is used when the server communicates directly with the client sending the response right after receiving the message from the client (within the acknowledgement message). The separate type of messaging is used when the server sends the response separately from the acknowledgement message that can take some time to be done. CoAP can communicate messages identical to HTTP CRUD messages in order to retrieve (GET), create (PUT), update (PUSH) and delete (DELETE) [34].

In GET request the client can set a special flag to enable a subscription feature in CoAP, after that the client will start receiving notifications that will be delivered at "best effort" eventually. As we mentioned earlier, CoAP provides reliability by confirming the messages (confirmable or non-confirmable types of messages) however since UDP has no embedded delivery control mechanism, CoAP can provide traffic overflow in case of sending data again and waiting for the acknowledgement message from the receiver [38].

CoAP supports usage of multicast packages that can be problematic for some types of networks. Moreover since the security layer in CoAP is provided by DTSL because that does not work with multicast messages used for group communication, it can cause the additional complexity for UDP when dealing with unreliable environments [14].

3.3 XMPP

Extensible Messaging Presence Protocol (XMPP) was designed to connect people with each other through instant communication for using in chats and messaging applications (Figure 3.4). Standardized by IETF around 10 years ago, it was being used in the Internet since then and proved its efficiency over the time. Main advantage of this protocol is human-readable addressing scheme name@domain.com that facilitates people connection in Internet.

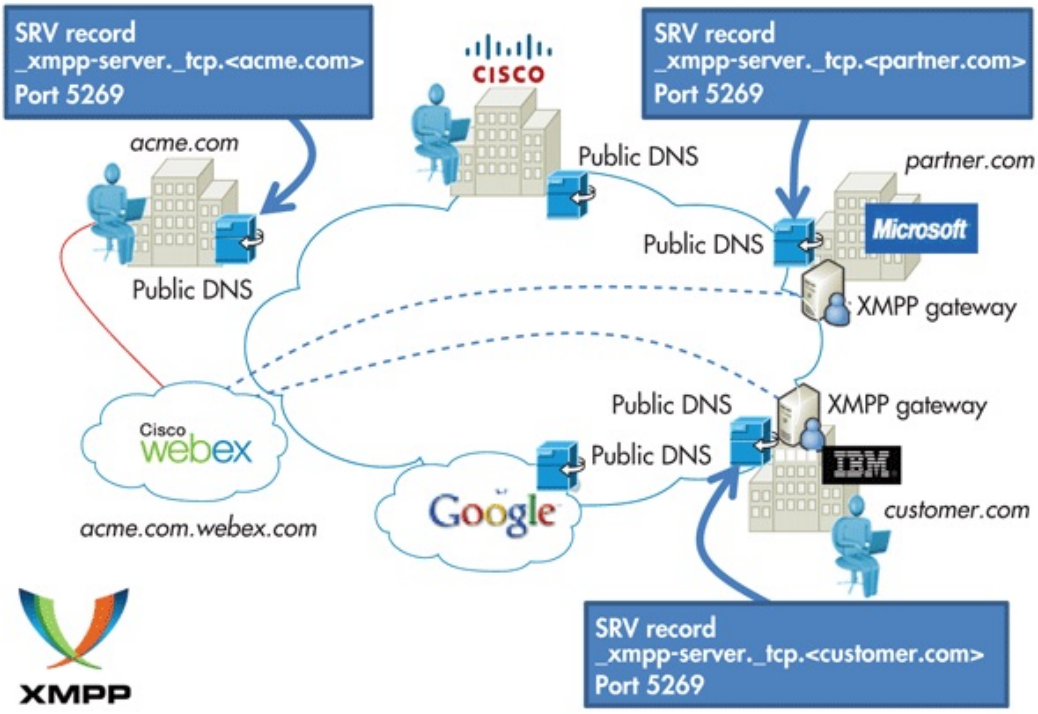

Figure 3.4: XMPP communications model [37]

It runs on top HTTP (therefore on top of TCP). recently it has been used for IoT applications, mainly because of usage of XML text format that makes the protocol easily extensible. XMPP provides the developers the choice of

desired architecture because it supports both publish/subscribe along with request/response architectures. It was designed for real-time systems and supports low-latency small messages.

XMPP has some shortages that becomes the reasons of relatively rare use for IoT applications. It does not provide the quality of service guarantees that makes it not suitable for device - to - device communication. As well as XML format with lots of headers and tags create additional overhead and increased power consumption that is not good for IoT applications [33]. XMPP provides reliable service but is quite slow. "Real time" in XMPP means from the point of view of a person measured in seconds, but not from device-to-device point of view [37].

3.4 AMQP

The Advanced Message Queueing Protocol (AMQP) is the protocol that is used mainly in financial industry to meet the need in reliable and secure delivery of important transactions. It operates over TCP and uses publish/subscribe architecture therefore it is similar to MQTT.

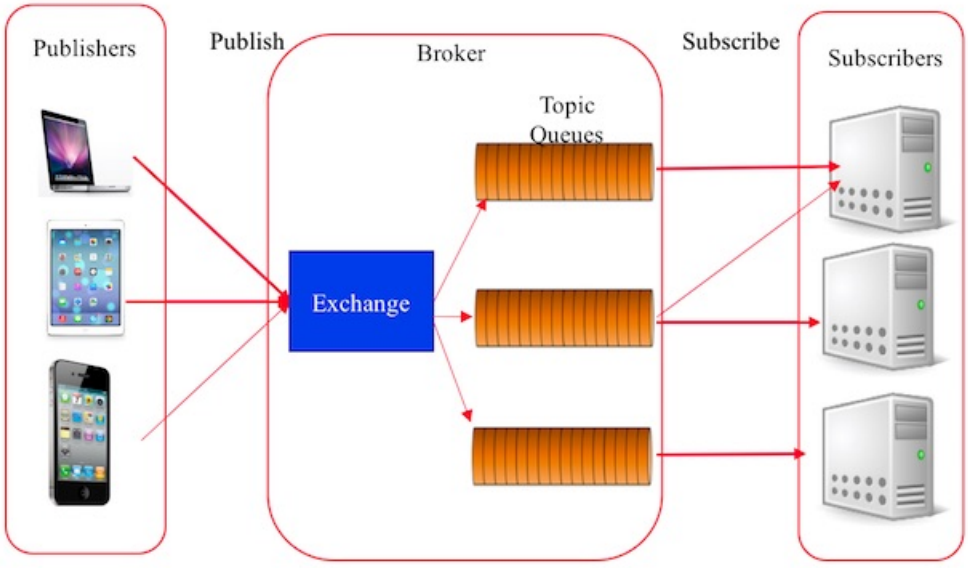

Figure 3.5: AMQP architecture [34]

The difference from the MQTT architecture is that in the broker layer there are two main parts (Figure 3.5) : a) exchange that is responsible for

receiving the messages from the publishers and sorting them to queues according to special conditions and roles specifically defined earlier, b) queues represent the specific themes and topics and collect the data that meets the specified topics. Queues are subscribed by subscribers that will get the data when it is available in the queue [34].

AMQP is a reliable protocol that provides quality of service guarantees as at-most-once, at-least-once and exactly-once delivery options [30]. This protocol is preferably used for server-to-server interaction for servers that can be deployed with different platforms to communicate for example, data necessary for analysis. The advantage of interoperability of AMQP fades away by heaviness of this protocol that make it hard to use on the lightweight IoT devices.

3.5 DDS

Data Distribution Service (DDS) is another protocol of publish-subscribe type that is used for device-to-device communication. It was designed by Object Management Group (OMG) for the purpose of distribution of data to another IoT devices. It has the benefits of high level quality of service (QoS) and high reliability because it does not have broker in its architecture. Therefore it does not have bottlenecks as other protocols that mainly rely on brokers in their operations [34]. It is suitable for IoT and device-to-device communication and can efficiently send millions of messages per second to many devices simultaneously.

It offers an opportunity to filter and select exactly which data goes to which receiver. Devices can be small or big, for small ones there is a lightweight version of DDS available that is suitable for running in a constrained environment. DDS was used in high-performance sectors such as defence, industrial and embedded applications such as military systems, wind farms, hospital integrated applications, medical imaging, asset-tracking systems, automotive test and safety applications[37]. DDS offers 23 QoS levels with criteria like security, urgency, priority, durability, reliability and more. It has two sublayers: publish/subscribe (which is data-centric) and reconstruction sublayers (which is data-local).

The first sublayer is responsible for the delivery of messages to the subscribers, the second one makes integration of DDS protocol to the application layer easier though it is optional. Data from the sensors is distributed by the publisher layers. It has two parts - data writer and data readers that basically take the responsibilities of the broker in the broker-based architectures. Data writer communicates with the publishers to decide which data

and which changes in this data will be sent to subscribers who are the receivers of the sensory data that is delivered to the IoT application. Data readers read all the data that is published and deliver it to the subscribers [34].

Data is communicated to devices from devices using relational data model. It is called "DataBus" communication and is a networking equivalent of a database. Data bus controls all the data updates and data access by thousands of simultaneous users the similar way a database controls access to the stored data. This feature of DDS is suitable and needed by the high-performance devices that work together as a unified system in complex real-time applications [37].

In this chapter we described different messaging standards that exist and are used right now in different IoT applications. These standards have different features and provide various opportunities of use. In the next chapter we will describe another two messaging standards for IoT developed by the Open Group - O-MI/O-DF.

Chapter 4

Open Group Messaging Protocols for IoT

The Open Group is a global consortium that is aimed to develop new IT standards in various areas including IoT. The IoT work group of the Open Group has the goal to develop a unique standard that will solve the interoperability problem in IoT. The Open Group standards will do for IoT that HTML/HTTP did for web - making everything in IoT connected on the fly.

The Open Group was working on and produced two messaging standards: The Open Data Format (O-DF) and Open Messaging Interface (O-MI). O-DF represents the information in a standardised way that is understandable and readable by the majority of information systems [42]. O-MI is an analogue of HTTP for the Web, as it can transmit the payloads between the devices and information systems in any format [43]. In this subchapter we will describe O-DF and O-MI more in detail.

4.1 O-DF

Open Data Format (O-DF) is a standard for representing payload for IoT applications. It was developed by the Open Group in order to represent information entities about different objects: devices, services, humans and so on. The representation of these objects in O-DF is general, independent from application or context. O-DF messages can be transported via various methods, the transportation itself is not part of O-DF standard, therefore O-DF data can be transmitted via network by different low-level network protocols or even manually by USB storage drive[42].

O-DF is specified using XML schema. It is designed to create information structures in a similar way they are created in object-oriented programming:

objects and properties of objects. It is general standard for representation of any object and it useful in such domains as IoT and lifecycle information management. In IoT there is a problem of publication of data from different sources such as devices, machines, servers. These devices must be able to publish their data, provide the opportunity for consumers to access the data, ensure the authenticity and integrity of the data using secure mechanisms and provide the opportunity to filter the information according to consumers' preferences: consumer identity, parameters, context [42].

A structure of O-DF message is a hierarchy with an Objects element on top of it. This element can contain any number of Object sub-elements. Each Object element usually has an id sub-element as an identification and optional description sub-element that provides additional information about the object for the users.

Object elements also have properties represented with InfoItem sub-element and any number of Object sub-elements. InfoItem sub-element can contain additional sub-element called MetaData which has name and values that represent values from the context of InfoItem. MetaData contains the description of InfoItem though the structure of it is similar to the structure of InfoItem. It can be useful to identify Infoitem in the situation when the system tries to retrieve unknown Infoitem. The example of InfoItem with multiple values can be seen in Figure 11. As we can see in the example of the O-DF element hierarchy (Figure 4.1), the Object tree can have any number of levels [42].

Because of hierarchical structure of O-DF, it can be used in RESTful services by sending the data using ordinary URL (Uniform Resource Locator) address using URL mapping technique [42].

This technique allows the client to request information by connecting to the special URL that includes designated object's id and other properties. As an example of HTTP GET request, the object can be accessed using the command **wget URL/REST/Objects** query. If the client wants to access only specific object, for example Refrigerator and get its current power consumption, it can be done using the following commands:

wget URL/REST/Objects/Refrigerator/id
wget URL/REST/Objects/Refrigerator/id/PowerConsumption

O-DF offers a way to create, manage and send the information about things in IoT in a standardized, understandable and universal way. Though we mentioned that O-DF can be transmitted using various methods, the main purpose of Open Group was to make it use particular transport protocol called O-MI (Open Messaging Interface) in a query/response format. In the next sub-chapter we will describe the O-MI standard [42].

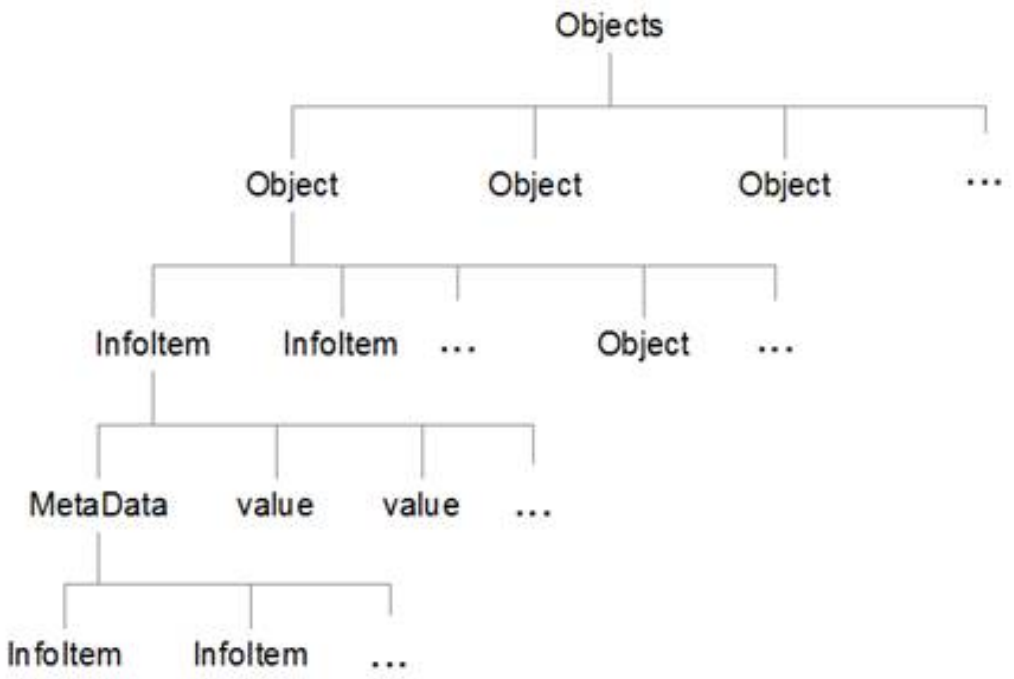

Figure 4.1: Illustration of O-DF element hierarchy [42]

4.2 O-MI

Open Messaging Interface (O-MI) was created by the Open Group by taking into example the HTTP and its role in the Internet. In the Internet information in HTML format is transmitted by the HTTP protocol into the browser for human use, and in IoT O-MI is used for transmitting data (preferably in O-DF format) for information systems use. IoT devices can use O-MI to communicate and interact with each other. The key of O-MI is in general approach to fit the majority of "things" in IoT that can be basically anything. As an example, in product lifecycle applications products use O-MI to communicate with distributed information systems that consume and publish information in real-time mode.

O-MI can be applied in the lifecycles of everything: people, services and so on [43]. O-MI can transmit data in any format, such as JSON, XML, CSV to any O-MI node across the network. O-MI nodes play the roles of client and server, they use peer-to-peer type of communication with other O-MI nodes or any other information systems and back-end servers [20]. Unlike other existing messaging standards that do not have an ability to suite the

requirements and needs of real-life closed-loop lifecycle management applications [19–21, 42] without any big modifications, O-MI standard was designed in the way to overcome problem and to have functional characteristics that can fulfill these requirements.

O-MI key characteristics [24]:

1. **Ability to use "low-level"** transportation protocols. Protocols include not only network protocols like HTTP, SOAP and SMTP but also mean of data transportation like copying on and from USB memory sticks, sending data in the messages on the mobile phones.

2. **O-MI supports three main operations: read, write and cancel.** *Read* operation is for retrieval of information from the things. It can be *immediate read*, in this case O-MI nodes can ask for particular values for the particular period of time, for example, for historical data or current data at the moment.

   Also O-MI supports *deferred read* also called as subscription. During subscription the read request is being sent with the specified interval rate and callback URL that is optional. After subscription has been made the requester will receive the designated data that will be sent to him using the earlier specified callback URL. If there is no callback URL, data can be accessed by read request with specific subscription ID that is contained in the response of the subscription request. This feature is useful when there are firewalls or in the situation when the NAT does not allow the response to reach the callback URL.

   Write operation is provided for sending data between the O-MI nodes and sending the updates at anytime. Cancel operation is used to end the subscription before it is expired.

3. **Different payload formats are allowed.** There is a deferred format for O-MI: O-DF (in XML), however O-MI can is able to send payload in any text-based format that can be included in the XML message. Return elements of O-MI response can also have data in other text-based formats.

4. **Time-to-Live (TTL) parameters have to be specified for requests and responses.** TTL is the time period during which the request is valid for transmission to the O-MI node. If the TTL is over the request has to be removed and the error message has to be sent instead.

5. **Nodes communicate synchronously between each other.** In any response message it is possible to include a new request message. This feature can be useful for some type of applications, for example, control applications. Also it makes possible for client to communicate with nodes that are located behind the firewalls and NATs.

6. **Publication and discovery of new services and metadata.** It allows data providers to publish their services using write operations as well as to discover services using search engines and URL mapping (by sending RESTful URL-based queries "HTTP GET").

7. **Specify the target O-MI node in request.** It means that the requester can list the specific O-MI nodes in the request and the O-MI nodes that receive the request that will have to re-send the response to the target O-MI nodes and will be responsible for possible error handling.

We see that O-MI is general tool used to send and deliver messages in every text-based format between the nodes. It offers read, write operations and also subscription options with opportunity to cancel it. Subscription can be tuned according to the demands of the requested: on interval basis, on update and on demand. Subscription feature is one of the main characteristics of O-MI standard.

In this chapter we described messaging standards provided by the Open Group. These standards along with several standards, described in the Chapter 3, are nowadays used in the majority of IoT applications wordlwide. But which standard is the most appropriate for IoT taking into account the requirements and demands of real-time systems? In the next chapter we will figure out the most appropriate messaging standard by comparing all of them with each other.

```
<omi:omiEnvelope ttl="1.0" version="1.0"
    xmlns="odf.xsd"
    xmlns:omi="omi.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <omi:response>
      <omi:result msgformat="odf">
        <omi:return returnCode="200"></omi:return>
          <omi:msg>
            <Objects>
              <Object>
                <id>SmartHouse</id>
                <Object>
                  <id>Kitchen</id>
                  <InfoItem name="Humidity">
                    <value unixTime="1459348542"
                    dateTime="2016-03-30T17:35:42.413+03:00"
                    type="xs:double">
                    0.5663739287794303
                    </value>
                  </InfoItem>
                </Object>
                <Object>
                  <id>WorkHouse</id>
                  <InfoItem name="FrontDoor">
                    <value unixTime="1460970721"
                    dateTime="2016-04-18T12:12:01.063+03:00"
                    type="xs:double">
                    30.967197215931222
                    </value>
                  </InfoItem>
                </Object>
              </Object>
            </Objects>
          </omi:msg>
      </omi:result>
    </omi:response>
</omi:omiEnvelope>
```

Figure 4.2: Infoitem with multiple values [42]

Chapter 5

Suitable Messaging Standard for IoT

In the previous two chapters we discussed six messaging standards . In this chapter we will found out which standard is the most appropriate for use in IoT applications.

There are several functional requirements for the IoT Messaging protocol, identified by Främling and Maharjan [11]. In addition there is a comparison framework for defining the concept of distributed object messaging by Tai and Rouvellou [39].

First, we will describe the requirements for messaging protocol that should be used in IoT applications and then introduce the comparison framework and its terms (Sub chapter 5.1). After that we will compare the protocols and define one that is the most useful in our case (Sub chapter 5.2).

5.1 Introduction to IoT messaging requirements and Comparison Framework

There are several main functional requirements that suitable for IoT messaging standard should have [10, 11, 24]:

1. Implementation for any kind of instances not depending on the application domain.

2. Implementation for variety of information systems (mobile, embedded systems etc).

3. Support for "synchronous" messaging such as immediate read and write operations, including "client-poll" subscriptions

4. Not restricted to one communication protocol only, it must allow sending messages using protocols like HTTP, SOAP, SMTP, as file copies
5. Possibility to create ad hoc, loosely-coupled, time-limited information flows "on the fly"
6. Peer-to-Peer communication possibility for all devices, i.e. client and server functionality can be implemented for any device, depending on available processing power, etc.
7. Handling mobility and intermittent network connectivity, i.e. support for asynchronous messaging capabilities that imply for instance message persistence, time-to-live, etc.
8. Context-dependent discovery of instances, instance-related services and meta-data about them
9. Context- and domain-specific ontologies are supported
10. Information of different kinds about more that one instance can be retrieved using queries by regular expressions.
11. Queries to retrieve historical data between two points in time.

These requirements are also reflected in the parameters of messaging classification framework, introduced by [39]. We will use this framework to compare existing IoT messaging protocols.

The classification framework is organized around three models: message delivery model, message processing framework, and message failure model. Each model represents the number of fundamental messaging features. And for each feature there are some variations.

5.1.1 Message Delivery Model

The message delivery model contains the features connected to the message exchange between message sender and message receiver. It is linked with notifications of receivers that some event (state transition) has occurred. Message delivery model has the following properties:

1. **Messaging API** property defines which types of operations are used for messaging. There are two types of operations: application-independent (describes generic operations for sending, receiving and administering the messages) and application-specific operation (interface of application functionality is defined by the developer of this application). As a variant a combination of these types of messaging API can be used.

2. **Initiation** defines how the message delivery happens. It can be initiated by the sender (server) who sends a message to the receiver (client) (*push initiation*). Or a receiver can send a query to the server in attempt to get a message (*pull initiation*). There also is a mixed initiation type when the same message can be pushed by the sender and pulled by different receivers.

3. **Intermediation** shows if there is any intermediate party in the message exchange. Intermediate party can be a message queue or channel objects.

4. **Persistence** property specifies if there is a data persistence feature in the messaging protocol. Message can be persistent in case it can be saved in the persistent store, that is important in case of system failures so the message can survive in it. Message can be transient. In this case message data is saved temporarily while the Time-To-Live (TTL) period is on.

5. **Subscription** defines if a receiver has the ability to subscribe to messages from intermediators or senders directly. Usually subscriptions are needed for push event notifications. Subscriptions can be time interval-based and event-based.

6. **Self-contained**: shows whether the message includes all necessary parameters and data to be understood by the receiver in the right context. For example, it shows if the message includes TTL parameter or indicated an operation that should be performed etc.

7. **Protocol agnostic** property shows if the messaging protocol can support low-level transportation protocols without necessity to change the messaging protocol itself. In protocol agnostic messaging protocol low-level transportation protocols such as HTTP, SMTP, SOAP can be used to transfer the data or even there is a variant to use physical transfer mechanism such as USB sticks.

8. **Synchronicity** defines the type of synchronization of message delivery from the sender to its recipients (end users or intermediators). The messaging model implies asynchronous communication between the sender and the ultimate recipients. However the synchronous communication via an intermediator can be implemented as an option to organize asynchronous communication.

9. **Delivery-guarantee** shows the level of guarantee that is provided for delivery of the message. There is three types of guarantee: low level guarantee (best-effort delivery) that is basically no guarantee of delivery, basic delivery guarantee (at-most-once) and the highest level of guarantee (exactly-once) that does or does not provide the visible acknowledgments of message delivery to the sender.

10. **Piggybacking** property shows if the protocol allows to send new request with the response. This property is important for real-time applications and applications that are fenced by firewall.

11. **Multiple payloads** property shows if the protocol supports different formats of data.

5.1.2 Message Processing Model

The message processing model contains characteristics of the process of communicating back the result after the message has been delivered. Message processing happens after the event notification and it is connected to the asynchronous request processing, that is requesting the result from the remote server asynchronously.

In asynchronous communication there are two essential roles for software components: the role of a *client* and the role of a *server*. A *client* sends the request message and consumes the reply message. A *server* consumes the request message and sends the reply message. There are two properties for message processing:

12. **Processing result** shows in which format the message is returned to the requester. There are three formats: a single return value, a single integrated return value, a set of individual return values. The first format is used when the server sends only a single response. The latter two are used when the same message is processed by many recipients that get back many acknowledgments of results and/or processing results.

    In order to integrate multiple returns the messaging protocol should have an intermediator. The single integrated return value format implies that the response will contain the requested value and the data. The third format is used when response is sent at different time intervals.

13. **Communication defines** how the result is received by the client. The client may receive a *separate reply message* to the request which will

be associated with the request by the id paired id (request/reply id). There are also two other approaches: *callback* approach and *polling* approach. When the callback approach is used the client sends request with a callback object reference. Then the server invokes it when the result is ready. When the polling approach is used the client queries the poller object for the results which is received by the client with the request previous.

5.1.3 Message Failure Model

The third category defines the protocol actions and rules linked to message delivery failures or other errors. In this category there is only one criterion:

14. **Failure Notification**: shows the reaction of protocol to the failures, specifically how it sends failure notifications. There are three types of notifications: timeout acknowledgments, reply with error message, and exception.

5.2 Comparison of Messaging Protocols

We will compare the five protocols that we described in the previous chapter with O-MI/O-DF using the comparison framework introduced earlier. The results are in Table 5.1. It is visible that AMQP and O-MI/O-DF cover the majority of properties and there is opportunity to choose between the options. All the protocols offer intermediate parties to support message transactions however only O-MI/O-DF protocol has the optional property of interval-based subscriptions that can be very useful for real-time IoT applications.

The property of self-contained message types is very important when in different systems the same field names can mean differently. Self-contained messages allow every system to better understand the semantics of the information contained in the message. AMQP and O-MI/O-DF have that property offering a possibility for processing software to load the field descriptions using namespaces and links. Only O-MI protocol is protocol agnostic and can use different low level transport protocols. Another protocol from the list - CoAP can be easily integrated with RESTful HTTP services but it runs on top of not reliable UDP protocol.

Piggybacking feature is supported only by CoAP, DDS and O-MI/O-DF. This feature is very important for IoT devices. For example in mobile dynamic environments it can be hard to provide a fixed IP address for every

device. Piggybacking function makes it available for devices without fixed IP address and located behind the firewall to carry out information exchange. In this case the feature of callback is very helpful. It can be used for subscriptions allowing to define the special URL where the data or events will be sent. So if there is no fixed IP address for a device, the processing point with the fixed IP can be addressed by a callback, and the data from different sources can be sent there so it can be processed further. O-MI/O-DF is the only protocol that provides this feature.

The feature of supporting different formats of payload is supported by all the protocols. This feature is important in IoT because different systems that to interact with each other have different standards. With this property combined with self-contained property different systems can better understand how to use the data that was received from the payload. Especially in the case when the system evolves and the number of interactive actors increase in many times.

Out of all protocols, CoAP and MQTT were designed to operate in local environments so they need to be wrapped into a websocket or modified in another way to be able to communicate on a global level.

As a result of comparison O-MI/O-DF standards is the most suitable standard for IoT applications. It provides reliable, interoperable and portable communication infrastructure for all kinds of devices and systems of devices.

Table 5.1: Messaging protocol comparison based on 14 criteria.

Property	Sub-property	MQTT	CoAP	XMPP	AMQP	DDS	O-MI
Messaging API	Application-specific		✔	✔			
	Application- independent	✔			✔	✔	✔
Initiation	Push	✔	✔	✔	✔	✔	✔
	Pull		✔	✔	✔	✔	✔
Intermediation		✔	✔	✔	✔		✔
Persistence	Transient	✔	✔		✔	✔	✔
	Persistent	✔	✔		✔	✔	✔
Subscription	Interval-based						✔
	Event-based	✔	✔	✔	✔	✔	✔
Self-contained					✔		✔
Protocol-agnostic							✔
Synchronicity	Synchronous	✔		✔	✔	✔	✔
	Asynchronous	✔	✔	✔	✔	✔	✔
Delivery-guarantee		✔	✔		✔	✔	✔
Piggybacking			✔			✔	✔
Multiple payloads		✔	✔	✔	✔	✔	✔
Processing Result	Single return value	✔	✔	✔	✔	✔	✔
	Single integrated return value	✔					✔
	Set of individual return value	✔	✔		✔		✔
Communication	Separate message	✔	✔	✔	✔	✔	✔
	Callback address					✔	✔
Failure Notification	Timeout of acknowledgement	✔	✔		✔	✔	✔
	Reply with error message	✔	✔	✔	✔		✔
	Exception				✔	✔	

Chapter 6

Case study and requirements

In the frameworks of the current thesis we will implement a mobile application for Android OS that is directly linked to bIoTope project and it's smart mobility case. In this chapter we will describe the project that owns smart mobility case and especially its part that is connected with EV parking and charging service in Helsinki region. The application will mainly be focused on the EV charging services but since the EV chargers are usually situated on the parking lots, it is hard to separate these two use cases. We will also outline the user story and requirements for the project that will later help to specify the requirements for the user application for searching and booking EV charging services that the author of the current thesis was working at (user interface of it).

Many system requirements are very domain specific related to the realities of EV charging industry and may challenge the existing vocabularies to some extent. We will discuss which standards and semantic models can be used in the case study and how can they cope with domain specific details of the requirements.

6.1 What is bIoTope smart mobility?

BIoTope project attempts to create an ecosystem for smart objects and implement a concept of system-of-systems (SoS) in the platform where it will gather existing services that use different smart objects connected with each other. The goal of the project is to make proof-of-concepts in different areas of smart city and show the feasibility of the SoS platform concept.

The case that we are interested in the current thesis is a part of Smart Mobility pilot project for Helsinki region. The smart mobility pilot includes all the services that can be in demand for the car drivers or offered by them:

finding the parking free suitable lots, finding available charging stations, effective route planning service, publishing the information about the car and making it available for renting out services.

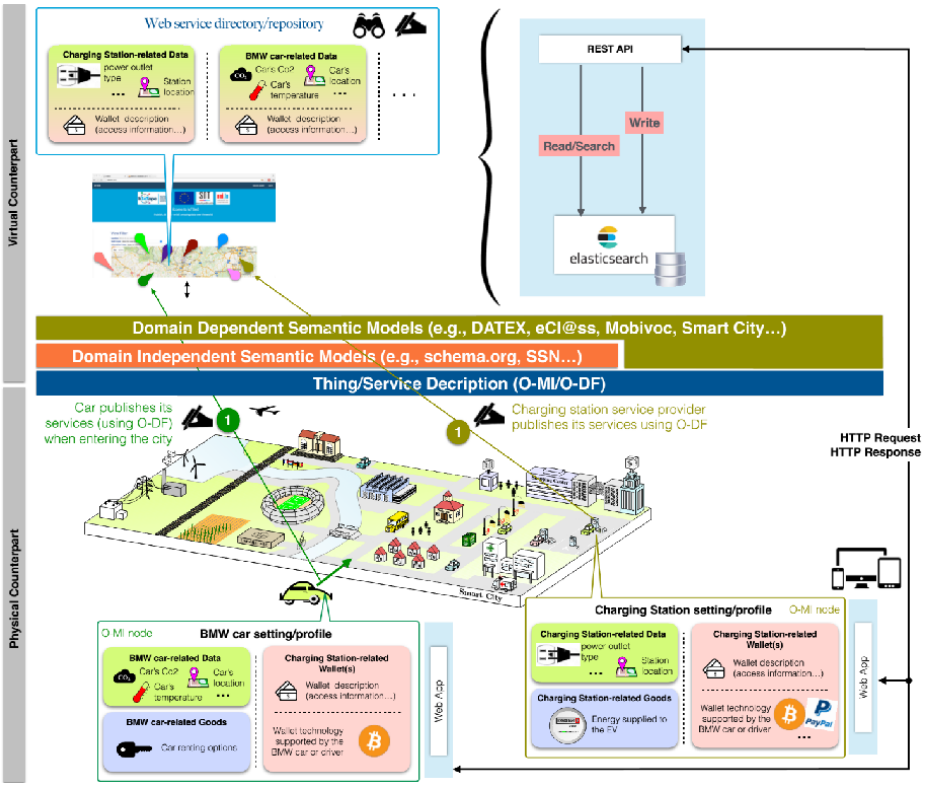

Figure 6.1: BIoTope big picture on the example of the Smart Mobility pilot case [4]

The Figure 6.1 above illustrates the structure of the ecosystem's platform. The platform is divided in 2 pieces: physical and virtual. Physical part depicts the car and the charging station that have real-time data (gathered and sent out by sensors) and static data that represents the description of the objects. Each object has services related to it: car-renting services offered by the car owner or charging service offered by the charger. All of the objects are organized as O-MI nodes. Their main goal is to publish the data related to the objects independently from their virtual representation.

The middle layers are a semantic filling of the objects information by the means of semantic models: independent or domain dependent. The higher layer - virtual layer is located in the web service repository. The communication will be organized via standard Internet protocols like HTTP.

There will be several APIs that will allow the business process to function around the objects [4].

6.2 EV charging use case and user story

In this thesis we will limit ourselves to the one case study that will focus in the smart EV charging services. In this section we will present the case study and its user requirements. The case study attempts to create a unified system for searching EV chargers along with an app for the users. The existence of one centralized system will lead to traffic and time waste reduction (in the process of searching and using charging services), higher trust from the user side towards service providers. The system will also solve the problem of different data sources for chargers and non- standardized interfaces of chargers data access.

The outcomes of the use case implementation will include easy and fast access to the reliable data about chargers, creation of a single interface and single data model for the whole service around the EU, the opportunity to easily update data about domain specific vocabulary with future development in vocabulary semantics.

The user is the owner of electric vehicle who enters the city by her car. She needs to recharge the car, therefore uses mobile app to search for charging stations. The user requests the list of chargers according to the desired destination or wants to look for the chargers nearby his current location.

The data about chargers is presented to the user via mobile app where she can select the charger on the map or in the list of chargers. The user is able to save her preferences for the charger also sort the search according to charger qualities and for the preferred location. The user sees whether the charger is available at the moment or occupied. The user picks preferred charger and books for the time period where it is available for use. Then user drives to the charger, identifies herself as a user who booked it and starts the charging service [5].

6.3 The system requirements

Among the main requirements of the system is the accessibility of data in real time database that contains reliable data about chargers and shows the availability of the charging points to the user [5].

The different types of data should be available in order to qualify and fit user decisions in the best possible way:

1. detailed location of the charger,
2. availability of the charger at the moment of search (reservation status),
3. availability of the booking system to reserve the charger in advance,
4. opportunity to see if the charger can be booked in advance or not,
5. the type of charger adaptor,
6. the speed of the charger,
7. the information about payment conditions that depend on the chargerâs provider rules.

These requirements should be reflected in the semantic data models of the system. In the following section we will discuss the available semantic models and chose the most suitable one according to the system requirements.

6.4 Semantic data models

Semantic data models will provide the semantic enrichment of the objects information. There are several semantic data models available: domain independent and domain dependent. We will explore the opportunities of each of the models and see how they comply to the requirements of the EV charging industry and its characteristics.

6.4.1 Schema.org

Schema.org is a collaborative community activity. Its purpose is to create and develop the usage of schemas for structured data on the Internet, in the web-sites, email messages and in other digital context. The vocabulary of this collaborative project can be used with many different encodings (JSON-LD, Microdata, RDFa etc).

The vocabulary contains the vast list of entities, relationships between them, their properties and actions. While the existing vocabulary covers many business domains and can be easily extended using the extension standard, it can be characterized rather as the domain independent semantic model.

Schema.org was founded and is being developed by reliable companies: Google, Microsoft, Yahoo and Yandex [36] and it has been used by vast amount of organizations and web developers. It is useful to explore its opportunities of application into the existing case study.

We analyzed the current schema.org vocabulary in the attempt to see which vocabulary terms will suit the requirements for EV Charging domain. After the closer look at the vocabulary we admit that schema.org can be characterized as domain independent semantic model. However some entities and their characteristics can be applied to EV chargers according to the required properties.

The closest domain entity to the EV charger is the GasStation [3]. It has properties from the Place entity and LocalBusiness entity. Some of the properties can be used for semantic modelling in the current case study: payment related properties (paymentAccepted, currenciesAccepted), business organization characteristics (openingHours, address, location, areaServed), place properties (geo, address, openingHoursSpecification), thing entity properties (name, image, description).

Another entity type that is related to EV charging service is Reservation and its more specific type ReservationPackage. Both of them have related properties that can be used in the EV charging project [36]:

- *bookingTime*: the date and time for the reservation;
- *programMembershipUsed*: this property can be used in the case when booking can be made only by the member of the EV charging provider program;
- *provider* : the service provider;
- *reservationFor*: reservation for EV Charging;
- *reservationId*;
- *reservationStatus*: the current status of the reservation including ReservationCancelled, ReservationConfirmed, ReservationHold, ReservationPending.
- *totalPrice*: the price for reservation;
- *underName*: the name of the person that reservation is for.

6.4.2 MobiVoc

MobiVoc initiative provides powerful vocabulary for modelling the mobility data. It intends to make the data communication between all the available data sources easier in order to answer the new challenges of the current smart mobility domain development [46].

There are the examples of these data sources that can be integrated with each other in one mobility service: map data, vehicle data, weather data, charging data, energy data, service description and others. MobiVoc provides standardised vocabulary using Semantic Web technologies and ontologies.

The vocabulary covers various mobility aspects using the recommended specification of World Wide Web Consortium (W3C) - Resource Description Framework (RDF) and lingua franca for integrating data and web [26].

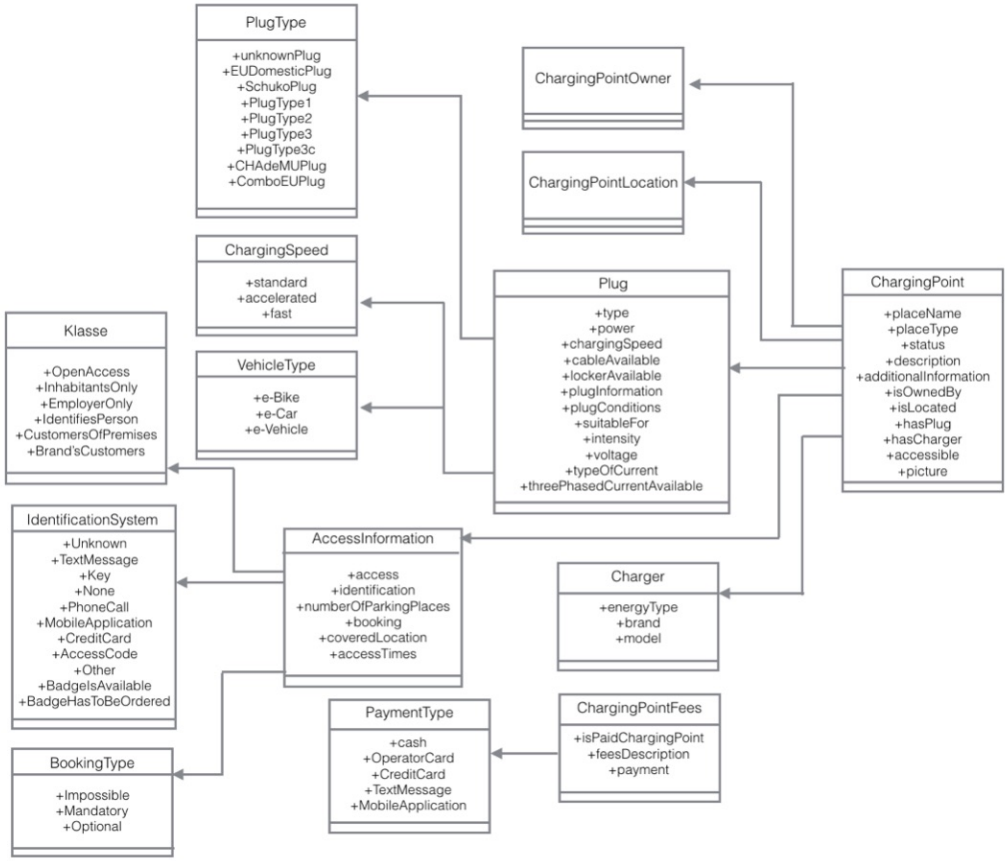

Figure 6.2: MobiVoc Schema [27]

After the exploration of MobiVoc types and entities we came to the conclusion that this domain-specific semantic vocabulary is more suitable for usage in EV charging system. MobiVoc was made specifically to reflect the mobility industry. MobiVoc contains several classes that are directly related to EV charging.

These classes are connected to each other and describe the properties and types of charging points, chargers, charging adaptors, charging fees and type

of access. The Figure 6.2 shows all the classes for EV charging [27].

In this chapter we provided an overview of the bIoTope project and its requirements. This project is a framework within which we implemented the software system for this thesis. The details of the implementation will be described in the next chapter.

Chapter 7

Implementation

The implementation part of the current thesis reflects the EV Parking and Charging system that was implemented as a part of bIoTope project and used for the current thesis. The author of the thesis implemented the user interface (Android mobile application) that consumes back-end Parking and Charging web service implemented by the group of programmers that are working in the bIoTope project. Though Parking and Charging service is not implemented by the author of the current thesis it is important to describe it in order to show the full picture of the system that has been developed. In this chapter we will describe the architecture of the system, implementation of the mobile application in details and make an overview the Parking and Charging service.

7.1 System overview

The EV Parking and Charging system consists of three main components (Figure 7.1):

1. **Client** that is represented by a mobile application for Android OS. This app was implemented by the author of the current thesis. The application is a tool for searching and booking the EV chargers in the designated area.

2. **Server** that has several layers and mainly consists of O-MI nodes. Server communicated with the mobile application using O-MI/O-DF messaging standard.

3. **EV Charger** built especially for the project, that is connected to the server side and opens or closes charging service according to the server calls.

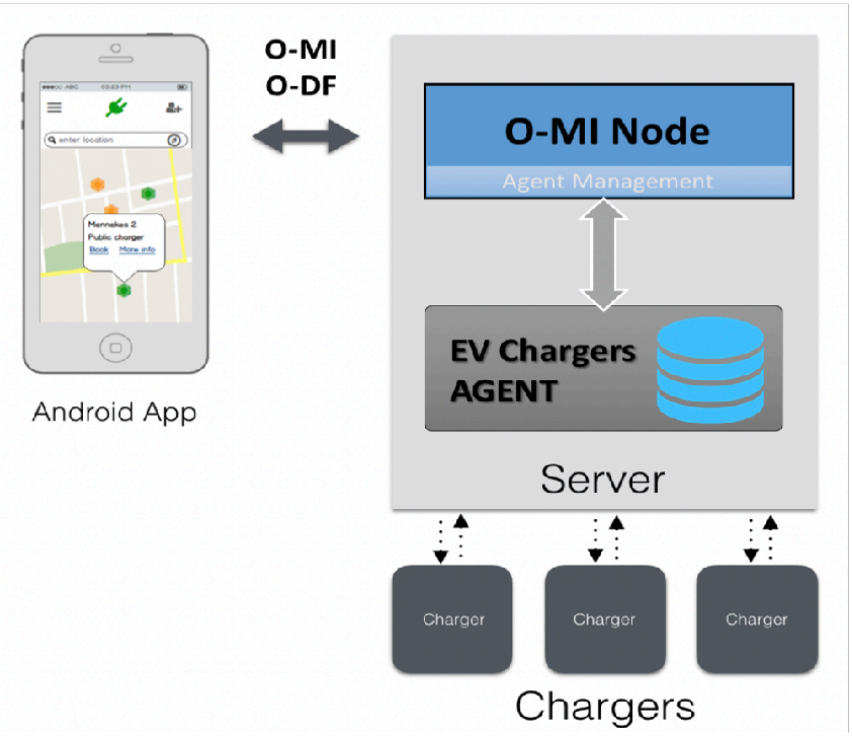

Figure 7.1: EV chargers and parking system overview [6]

The general system functionality can be illustrated by these points (Figure 7.2:

1. Charger creates event subscription to Parking Service for changes in Pole's parking space O-DF structure. Service answers with Response containing returnCode="200".

2. An electric vehicle enters Helsinki city and the user sends the geographical query to the mobile application to search for a parking lots with available EV chargers. App sends call request to Parking Service for method FindParking with information of destination and type of vehicle, etc. The Parking and Charging Service (the O-MI node) returns the list of parking lots with EV chargers.

3. The user can choose the particular parking spot with an EV charger, navigate to it (the route towards the EV parking spot destination can be defined using Google Maps automatically from the app). The user

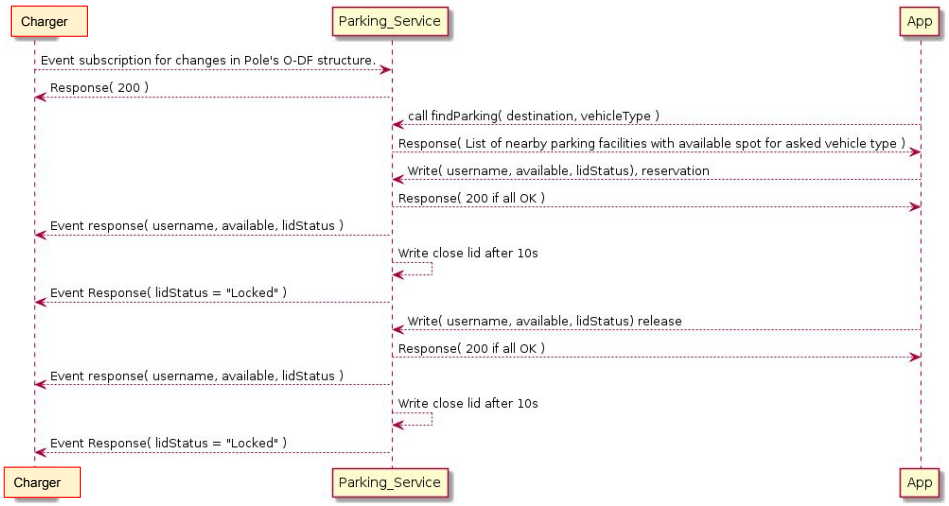

Figure 7.2: EV chargers and parking sequence diagram [6]

can reserve EV parking spot for use, sending the write request to Parking Service with user's name and picked parking spot set to "false" (if it is available at the moment).If Parking Service approves the reservation it answers with response containing returnCode="200".

4. After the parking spot has been booked the user can authorize himself to open the EV charger. The app sends the write request with user's name and lid status "open". The service returns the response with returnCode="200".

5. Because the information changed in O-DF structure of the Charger, a response is send to the Charger. This response contains only the changed values of the structure. Charger opens lid and turns power on so that charging is possible. User connects their cable to Charger and uses the charging service.

6. Parking Service changes Charger's lid status to "Locked" after 10 seconds after setting status to "Open". Causes a response to be send to Charger because O-DF changed again.

7. Pole activates lock so that lid will be locked when closed. User closes the lid. User is about to leave with their vehicle and uses the application to

release (unbook) the parking space and send releasing write to Parking Service.

In the following subchapters we will separately describe the user interface of the service (an Android application) and the back-end part of the service (server and EV charger).

7.2 User Interface Part: Android application

On the client side there is a mobile application for Android OS. The application was developed by the author of this thesis using Java language. The application can be run on a minimum Android SDK version 14 and maximum on Android SDK version 25 (Android Nougat).

The application was developed since January until June. There were two versions of the application:

1. **The first version** was developed in the period of time from January till March 2017. It contained simple user interface with map used for searching for the EV chargers using the current location of the user. The server part was simulated since it took time to implement the O-MI node and implement the physical EV charger.

2. **The second version** was developed during March-June 2017 when the server implementation was ongoing. The user interface allows the user not only to search for the EV chargers location according user's current location but also to use Google Map search for EV chargers in desired future location. The user can also reserve the EV parking spot and unlock the EV charger for immediate usage.

7.2.1 Application Use Case

The application represents simple and usable interface. The user of the app is able:

- **To search for the parking spots with EV chargers nearby his/her current location.** The launch screen of the app shows Google Map. There is an icon-button in the top right corner of the screen that enables EV parking spots search by the user's current location (Figure 7.3, screenshot 1).

- **To search for the parking spots with EV chargers in desired location.** In this case the user can use the search widget by typing the

name of the location in. The search widget will provide the autocomplete function provided by Google Places API.

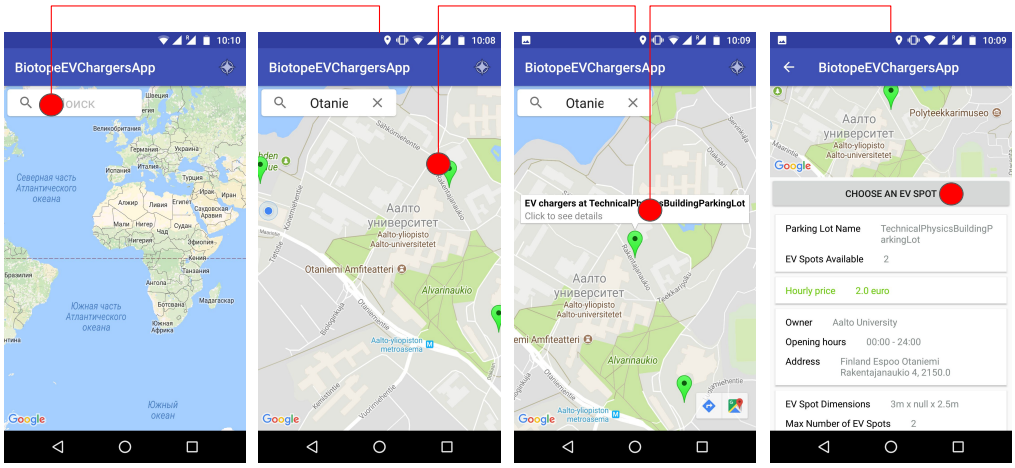

Figure 7.3: Using the app: Searching for EV parking spots

- **To see the search results pinned on the Google Map.** Search results show the parking lots that have parking spots equipped with EV chargers. The color of the pin reflects the current availability of the EV parking spots with the chargers. If the pin is green, it means that there are available parking spots with EV chargers in this parking lot. If the pin is red, there are no currently avvailable EV parking spots (Figure 7.3, screenshots 2,3).

- **To read the information about the parking lot**: availability of EV chargers, amount of free parking spots with EV chargers, opening hours, hourly price for charging and other related information (Figure 7.3, screenshot 4)

- **To read characteristics and to choose the desired EV parking spot** from the list of available spots on the parking lot.

- **To reserve parking spot with EV charger** for immediate use or to leave parking spot (Figure 7.4, screenshots 6, 8).

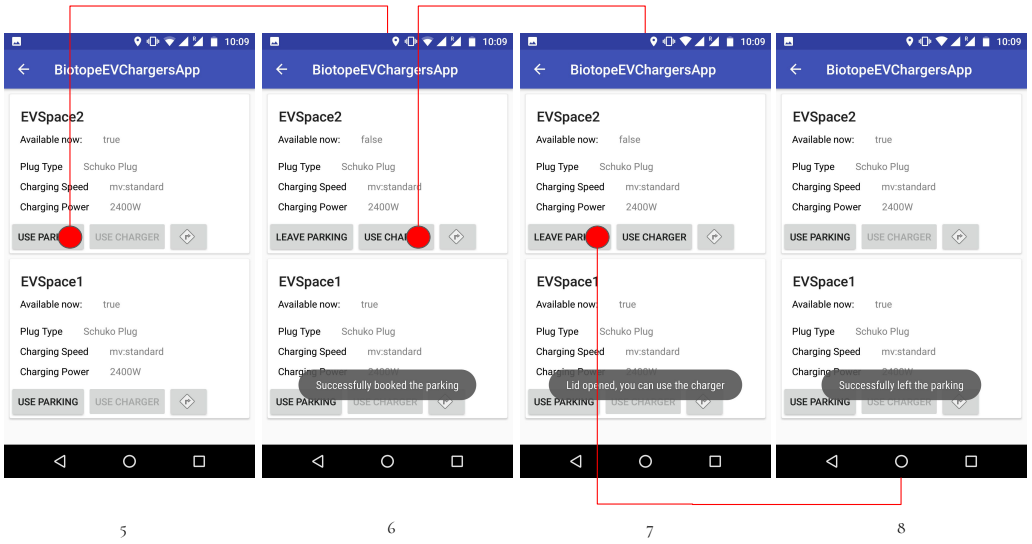

Figure 7.4: Using the app: Booking the parking spot and using the EV charger

- **To use the charger**: unlock the lid of the charger for immediate use of charging service (Figure 7.4, screenshot 7). The button "Use Charger" unlocks the charger's lid so the plug can be using inside the charger pole.

- **To get the directions to the parking lot** from the current location of the user by calling Google Maps from the application and loading the directed route.

7.2.2 Overview of the App Architecture

The architecture of the application is illustrated by the Figure 7.5. The application can be divided into three parts. Each part is responsible for its own tasks.

User Interface part constructs user interface and uses different asynchronous tasks (AsyncTask) to get the necessary data from the server. *API client* is responsible for communicating with the server, when it is activated by AsyncTasks. All the data that is received from the server is parsed and mapped to the model objects and supplied to the *User Interface*. We will describe each part of the app architecture in details in the following paragraph in order to explain how the app is supposed to operate.

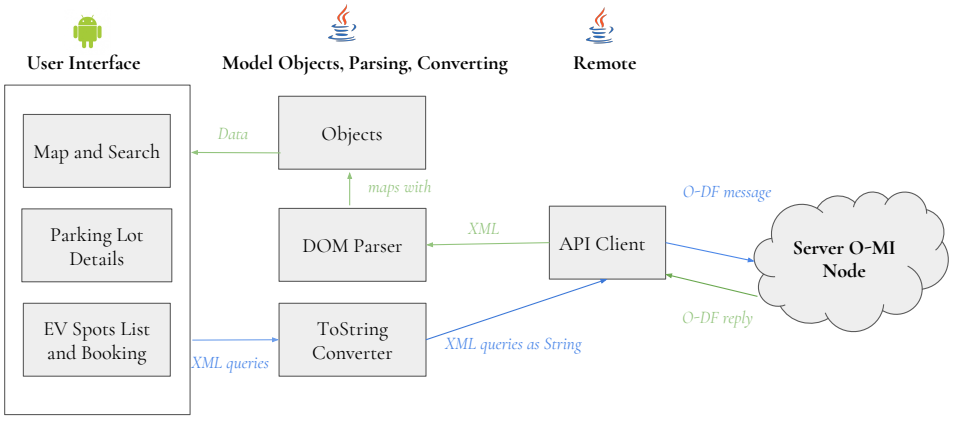

Figure 7.5: App Architecture

7.2.3 User Interface

User Interface part is responsible for user interface construction of the application, for sending the server requests in order to get the data from the server and show it to the user. It includes files that are specific for Android OS framework. User interface can be divided in three levels:

1. **Map and Search** that is the launcher screen and shows the first after the app has been launched (Figure 7.6). This screen is responsible for searching and getting the parking lots with EV chargers on the map. This user interface is being managed by the *ParkingMapActivity* that holds two fragments: *SupportPlaceAutocompleteFragment* and *ParkingMapFragment*.

   SupportPlaceAutocompleteFragment is responsible for searching the locations with autocomplete function.

   ParkingMapFragment is responsible for showing the map and for pinning the found parking lots to the map (putting the markers according to the location). It holds the AsyncTask (SearchParkingTask) that searches for the parking lots in the server using the geo location data.

   Geo Location are latitude and longitude of either the current user location or of the location of the desired user location in the future (the

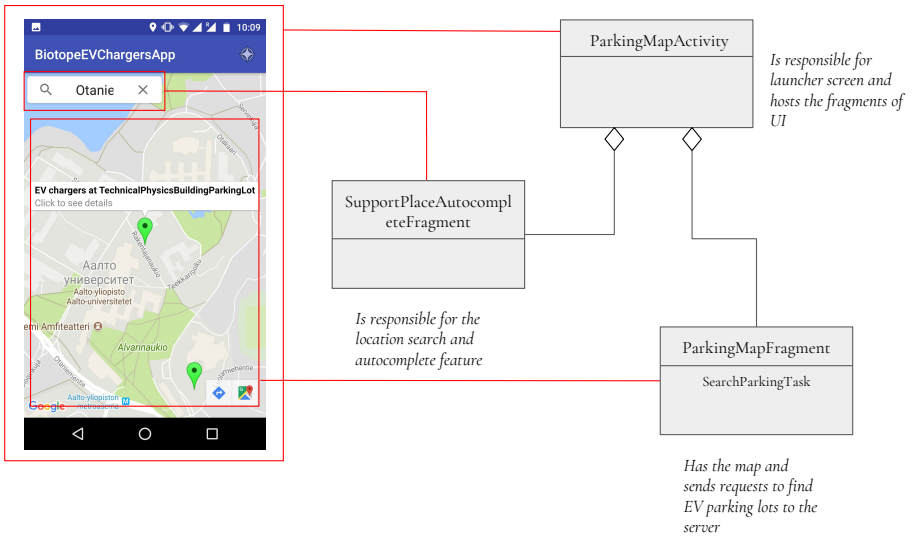

Figure 7.6: UI and class diagram of the launcher screen

place where the user is heading to). The desired user location latitude and longitude coordinates are supplied by the *SupportPlaceAutocompleteFragment*.

2. **Parking Lot Details Screen** (Figure 7.7) shows the characteristics of the parking lot that the user chooses from the map. This screen is operated by the *ParkingDetailActivity*. It has the map where the current parking lot is pinned to. It shows the information about the opening hours, address of the parking lot and other important information. It also has the button "Choose an EV Spot" that will bring the user to the next screen of the third level. On button click the AsyncTask (SearchParkingLotsListTask) is called that connects the server in order to get the renewed data about the EV parking spots that are available at the current parking lot.

3. **EV Spots List and Charger** (Figure 7.4) show the list of EV spots that are available on the chosen parking lot. From this list the user can reserve an EV parking spot and use the charger. The user interface is organized by the *EvSpotsListActivity* that hosts the list of the available EV parking spots. It has two AsyncTasks for making the reservation of the EV parking spot (*ReserveParkingTask*) and for opening the lid of the charger to use it (*UseChargerTask*). Each item (an EV park-

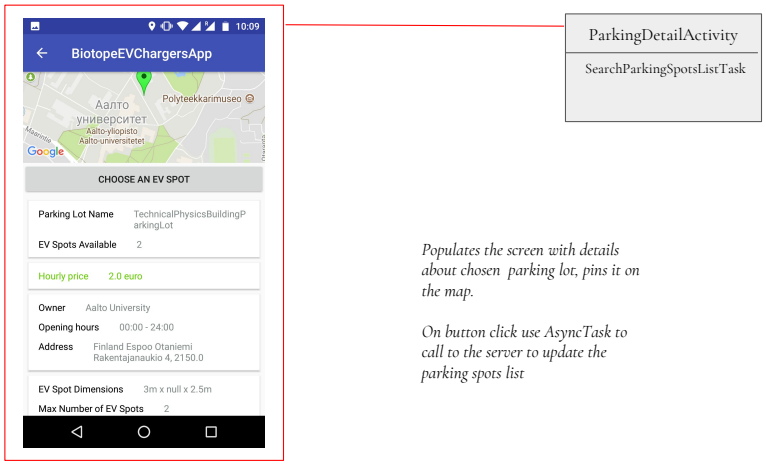

Figure 7.7: UI and class diagram of the parking lot detail screen

ing spot) in the list is organized by the *EvParkingSpotsAdapter* that manages the buttons for reservation of the parking lot and for using the EV charger. It also manages the information allocation in the list item: details of the charger, type of charging plug and so on.

7.2.4 Objects and data parsing

This part in the app architecture is responsible for data parsing, processing and mapping to the objects.

ToStringConverterFactory is responsible for converting XML queries to String object in order to pass this String object to the API client in the POST request.

XML Parser (DOM parser) is responsible for parsing the XML data from O-DF reply from the server and for mapping the data from O-DF elements to the related objects. Figure 7.9 shows the class diagram that includes all the objects (POJO) that exist in the application. These objects are constructed in the same way as the O-DF Object elements and the member variables inside the POJO objects represent InfoItems. Objects also can contain the list of other objects.

The main object element is *ParkingService*. It contains all other nested elements. According to the O-DF structure, *ParkingService* has the list of

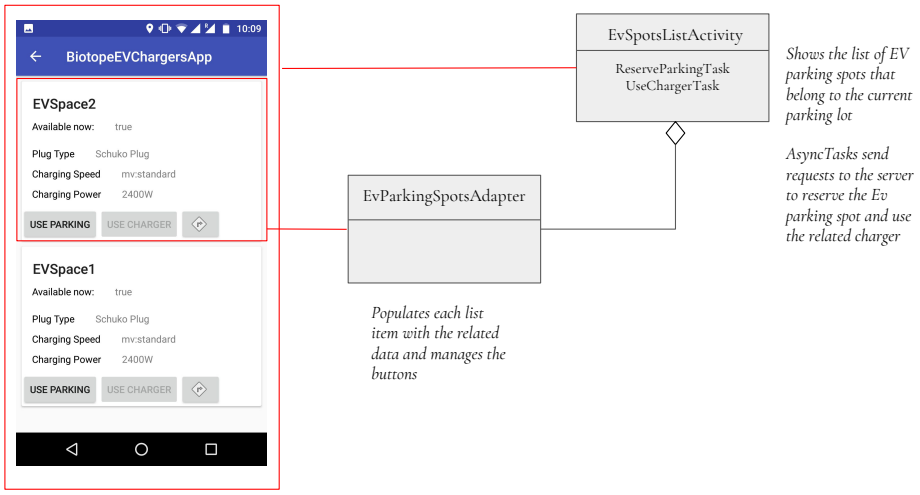

Figure 7.8: UI and class diagram of the EV spots list screen

object elements called *"ParkingFacilities"* (Figure A.1). In the application POJO named *ParkingService* has the list of *ParkingLot* objects. In O-DF structure *ParkingLot* object has several properties and contains the list of *ParkingSpaceTypes* (Figure A.2). In application there is a POJO *ParkingSection* that is an equivalent of *ParkingSpaceTypes* element in O-DF. The application only requests and uses *ParkingSections* with the specific id *"ElectricVehicleParkingSpace"*. But the *Parking and Charging Service* contains other types that can be used in other web and mobile applications (for example, for parking sections for motorbikes or bicycles or usual cars that do not need an EV charger).

In O-DF structure, each *ParkingSpaceType* has a list of *Spaces* (Figure A.2). In application *ParkingSection* has the list of *ParkingSpot* objects. Each parking spot that belongs to the *ParkingSection* designated to EV cars has a Charger object(Figure A.3). All in all the application uses the POJO objects that can be directly mapped to the O-DF structure of the Parking Service. The Figure B.1 compares all the names of the O-DF Object elements to the related POJO objects used in the application.

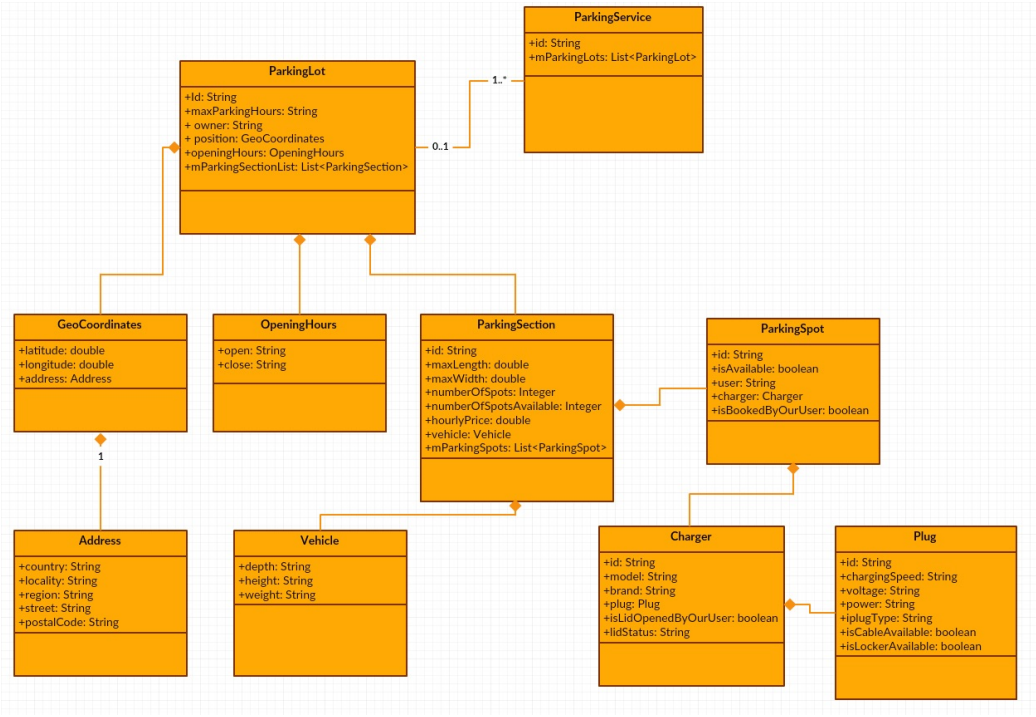

Figure 7.9: Class diagram of the objects

7.2.5 API client

API client is constructed using special library Retrofit. API client creates the *RetrofitService* that calls the O-MI node to the following API address **http:biotope.cs.hut.fi/omi/node/**.

RetrofitService makes the POST calls to the server adding the String object in the body of the call. String object contains the related query that is initially in XML format but it is converted to String by *ToStringConverterFactory*.

The app calls to the server using four different queries.

1. **Find Parking** call request is used to get the list of parking lots at the specified geo location (Figure C.1). In this query it is important to specify which type of parking lot it is needed. Since in the app we are interested only in the parking with EV chargers, the value of InfoItem *ParkingUsageType* should be set to *ElectricVehicleParkingSpace*. It is important to pass the coordinates to the value of InfoItems *Longitude* and /textitLatitude accordingly.

2. **Reservation of Parking Spot** write request is used when the user

wants to use the specific chosen parking spot (Figure C.2). In the request it is important to include the id of the parking lot and id of the parking spot. The value of parking spot InfoItem *Available* should be set to *false*. The username of the application current user should also be included in the request in the InfoItem *User*'s value.

3. **Unbooking of Parking Spot** is a write request that is used to free the parking spot after it has been used by the user (Figure C.3). The write call is similar to the previous one (to reserve the parking spot), however the value of InfoItem *Available* should be set to *true*.

4. **Open the Charger Lid** request is sent when the user wants to use the charger (Figure C.4). This request can be combined with the reservation request or can be sent separately after the parking spot is reserved by the user. In the request it is important that the parking spot's InfoItem "Available" should be set to false and the value of InfoItem *User* should have the username of the current user. The InfoItem's *LidStatus* value should be set to *Open*. In this case the lid will be opened and the user can start using the charger.

7.3 Back-end Part: O-MI Node and EV Charger

In this subchapter we will describe what the service back-end part contains and how it communicates with the client to let the reader understand the whole organization of the system. The server part for the EV Charging application was implemented by a team of programmers for the bIoTope project. The server part is organized over several components that communicate with each other. It consists of *software block* and *hardware block* (Figure 7.10).

Software block represents the O-MI node server that contains several components.

All O-MI basic operation are implemented by the server. The current underlying transport protocols that are supported by the O-MI node are HTTP and websockets. Any O-MI operation is transported using an HTTP POST.

O-MI service. The server has the element that is responsible for parsing the requests, authentication and authorization of the client. Then each request is handled by the Request Handler.

Request Handler sends the particular request to the next responsible elements according to the nature of the request. All the "read" requests are sent to the database, that is maintained by the server.

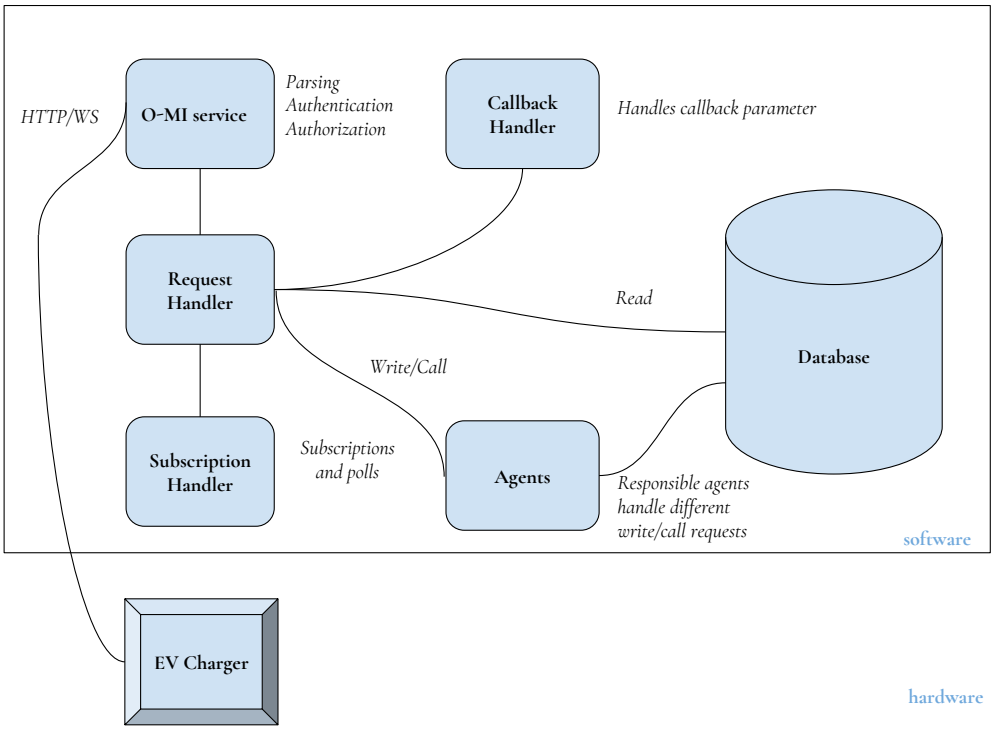

Figure 7.10: Organization of the back-end part

Database stores the data about O-DF structure including Object(s) and InfoItems.

All the requests concerning write/call operations are distributed by the Request Handler to responsible agents.

Agents. Agents are the actors that can programmatically interact with the core of an O-MI node. Generally an agent is a worker thread that gets data from different data sources separately. For that it uses specific protocols and turn them into objects that can be used by the O-MI node core.

Current implementation has one agent that communicates with the database and make the necessary changes in the data inside the database according to the "write/call requests".

Subscription Handler. There is a subscription mechanism offered by the O-MI server that allows responses based on the events or time intervals to a number of subscribers. In the current system there is only one subscriber to the O-MI server. It is the external EV charger - a hardware element of the system that aims at receiving the notifications about the lid status. All

the subscriptions and polls are handled by the Subscription Handler that receives related requests from the Request Handler. Subscription handler is also responsible for polling the data using subscription ID integrated into the read request (in case when callback address is not provided). In order to eliminate the active subscriptions Subscription Handler processes "cancel" requests that stop and delete related subscriptions.

Callback Handler O-MI server provides the subscribers a feature of callback, when the subscriber can specify a different URL from the address that has sent the request. It is called the callback address. So the message will be sent to the callback address during an active subscription. For handling these messages forwarded to a callback address there is an element called Callback Handler. In the current implementation the EV charger provides the callback address.

Hardware block includes the EV charger with the lid. EV charger hosts an ESP8266 Wi-Fi chip for control, A7 GSM module for long-range cellular communications and Sonoff relay for electricity control. The software has been written in C with very small memory footprint, and communicates to O-MI service using websockets.

EV charger is connected to the software block and receives the information about the charger lid status from the O-MI node. Since it is too difficult to maintain connection to the EV charger through GSM, the charger is subscribed to the messages from the O-MI node. The charger can operate the lid according to the lid status, communicated by the O-MI node.

The default status of the lid is "closed", so that the lid could be locked and the charger cannot be used. But If the lid status is "open" the charger unlocks the lid so the user can open it, put the charging plug inside and start charging. The status "open" is only valid during 2 seconds for security reasons. This period of time will be enough for the user to be able to open the lid and start using the charger. If the status changes to "closed", the lid automatically locks. However through the application the status can be easily changes again to "open".

Chapter 8

Conclusions

8.1 Summary of Findings

The main focus of this thesis research was to develop a mobile application using the O-MI/O-DF messaging protocols for IoT and test them in the smart EV charging context provided by the European Smart City project bIoTope. The first chapters of the thesis (chapters from 2 to 5) provide an overview of IoT concept, its ecosystem and issues, highlighting the necessity to create a unified standard for IoT messaging. Several existing messaging protocols were discussed and compared with O-MI/O-DF using the special comparison framework that reflects the main functional requirements for a general IoT messaging standard. This comparison showed that currently used messaging standards suit the requirements for unified IoT messaging standard only partially. However, O-MI/O-DF is the best candidate to implement the majority of these requirements (transport protocol independency, subscription, support of different payload formats and more).

Then, in chapter 6 the context of the thesis was introduced. The context was provided by the European Smart City project, bIoTope that aims to create a system of systems for IoT where many services from different providers can be published using the unified messaging protocol O-MI/O-DF. The project provided the system requirements for the implementation part of the current thesis research.

In chapter 7, we described the implementation part of the mobile application for smart EV charging. We started from the more specific requirements and use case of the application, then followed with the description of the system architecture and details of construction of the user interface for Android OS that communicates with the server part using O-MI and O-DF. We explained each part of the application and their functionality. We demonstrated

the queries in O-DF that mobile app sent to the O-MI node and showed the connection of the O-DF structure elements to the classes of objects that play key role in the application. O-DF was structured specifically for the use case of the application - smart EV charging. The mobile user interface was implemented by the author of the current thesis. In the end we shortly described the back-end part of the system (the O-MI node and charger pole) that was implemented by other group of programmers who participate in the bIoTope project.

8.2 Implications of Research

This thesis had three main objectives. The first one was to demonstrate O-MI and O-DF standards and to show why these standards are the most suitable standards for IoT messaging. The second objective was to develop a mobile application that communicates with O-MI node using O-DF messages. The third objective was to test the application in the context of smart EV charging provided by the European project bIoTope. The mobile application was implemented and it was demonstrated how O-MI and O-DF can be used in a mobile application.

8.3 Reliability and Validity of the Research

The mobile application that was implemented for this research thesis is just a proof-of-concept, therefore the application has a restricted functionality. The data about the EV chargers is mocked on the server that communicated with the mobile interface. However it uses the real charger pole constructed by one of the programmers specifically for the bIoTope project. The core design of the application, user requirements (that were provided by the bIoTope project) can be applicable to other systems. It can be reused in the real EV charging system with the actual data (existing parking lots and EV chargers).

8.4 Future work

We learned that O-MI/O-DF standards provide better interoperability for IoT than other existing messaging standards. It is particularly important for smart city applications. The result of this thesis is a mobile application that uses O-MI/O-DF messaging standards to communicate with the server. This simple application is an example that shows a potentially opened IoT

system in smart city context. A system that uses unified standard to publish and communicate different types of services from different domains.

However there are many ways to improve the mobile application and expand the service.

First of all, user profiles and user authentication feature should be created. The existing application is designed to be used only by a unique user. But for the full service capability it should support the features of registration, user account creation and authentication.

Then, when the user will be using the EV charger, the amount of the charged energy and the time should be also registered in the app.

The app should offer to a user an opportunity to pay for the charging service, so payment features should be implemented in the application. There are various solutions for digital payment offered by some providers such as PayPal and some others. These solutions could be integrated in the application in order not to spend internal resources for initial development of a payment solution from scratch.

The application could also add a feature of indoor mapping of the location of the EV chargers inside the big parking lots. It should make the chargers search easier for the user that arrived to the parking lot and is looking for the exact location of the parking space that he is interested in.

On the server side, the real existing parking lots with EV chargers should be published to the O-MI node. The server could contain other digital services that could be also published using O-MI and O-DF protocols. The parking lots could offer parking spaces for other kinds of vehicles such as motorbikes, ordinary cars, bigger vehicles.

The usage of O-MI/O-DF protocols opens a lot of opportunities for digital service providers in the smart city.

Bibliography

[1] Atzori, L., Iera, A., and Morabito, G. The Internet of Things: A Survey. *Computer Networks 54*, 15 (2010), 2787–2805.

[2] Baran, P. Reliable digital communications using unreliable network re- peater nodes. *RAND Corporation papers, document P-1995* (1960).

[3] Bellavista, P., Cardone, G., Corradi, A., and Foschini, L. Convergence of MANET and WSN in IoT urban scenarios. *IEEE Sensors Journal 13*, 10 (2013), 3558 – 3567.

[4] BIoTope Documentation. D2.1 ecosystem stakeholder requirements report and pilot definition, 2016.

[5] BIoTope Documentation. D2.4 biotope sos reference platform specification v.1.0, 2016.

[6] BIoTope Documentation. D6.2 proof-of-concept "smart building and equipment" implementation v1, 2017.

[7] Braden, R. T. A server host system on the arpanet. *ACM New York, USA* (1977).

[8] Cuff, D., Hansen, M., and J., K. Urban sensing: Out of the woods. *Communications of the ACM 51*, 3 (2008), 24 – 33.

[9] Främling, K., Holmström, J., Ala-Risku, T., and Kärkkainen, M. Product agents for handling information about physical objects. *Report of Laboratory of Information Processing Science series B, TKO-B 153/03, Helsinki University of Technology* (2003), 20.

[10] Främling, K., Kubler, S., and Buda, A. Universal Messaging Standards for the IoT from a Lifecycle Management Perspective. *Lournal of LaTeX Class Files 11*, 4 (2012).

[11] Främling, K., and Maharjan, M. Standardized communication between intelligent products for the IoT. *Proc. 11th IFAC Workshop on Intelligent Manufacturing Systems* (2013), 157 – 162.

[12] Gershenfeld, N., Krikorian, R., and Cohen, D. The Internet of Things. *Scientific American 291*, 4 (2004), 76–81.

[13] Hafner, K. Where wizards stay up late: The origins of the internet. *Journal of Engineering Education, Simon Schuster* (1998).

[14] Hartke, K. Observing resources in the constrained application protocol (coap). *RFC 7641* (2015).

[15] Hauben, R. From the arpanet to the internet. *A Study of the ARPANET TCP/IP Digest and of the Role of Online Communication in the Transi- tion from the ARPANET to the Internet* (2001).

[16] Hernandez-Muñoz, J., Vercher, J., Muñoz, L., and Galache, J. Smart Cities at the forefront of the future Internet. *The Future Internet, Lecture Notes Computer Science 6656* (2011), 447 – 462.

[17] Huvio, E., Grönvall, J., and Främling, K. Tracking and tracing parcels using a distributed computing approach. *In: SOLEM, Olav (ed.) Proceedings of the 14th Annual Conference for Nordic Researchers in Logistics (NOFOMA'2002) , Trondheim, Norway, 12-14 June* (2002), 29 – 43.

[18] Jankowski, S., Covello, J., Bellini, H., Ritchie, J., and Costa, D. The internet of things: making sense of the next megatrend, 2014.

[19] Jun, H. B., Shin, J. H., Kiritsis, D., and Xirouchakis, P. System architecture for closed-loop plm. *International Journal of Computer Integrated Manufacturing 20*, 7 (2007), 684–698.

[20] Kiritsis, D., Bufardi, A., and Xirouchakis, P. Research issues on product lifecycle management and information tracking using smart embedded systems. . *Advanced Engineering Informatics 17* (2003), 189–201.

[21] Kiritsis, D., and RolstadaÌs, A. Promise-a closed-loop product lifecycle management approach. . *in IFIP 5.7 Advances in Production Management Systems: Modelling and Implementing the Integrated Enterprise* (2005).

[22] Kortuem, G., Kawsar, F., Sundramoorthy, V., and Fitton, D. Smart objects as building blocks for the internet of things. *IEEE Internet Computing 14*, 1 (2010), 44–51.

[23] Korzun, D., Kashevnik, A., and Balandin, S.I.and Smirnov, A. The Smart-M3 Platform Experience of Smart Space Application Development for Internet of Things. *NEW2AN/ruSMART 2015, LNCS 9247* (2015), 56–67.

[24] Kubler, S., Främling, K., and Buda, A. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervasive and Mobile Computing 20* (2015), 100–114.

[25] Kurose, J. F., and Ross, K. W. *Computer Networking: A Top-Down Approach (5th ed.).* Boston, MA: Pearson Education., 2010.

[26] MobiVoc Website, Mission Statement. Supporting human mobility by data mobility. webpage, 2017. `https://www.mobivoc.org/en/goals.html`. Accessed 02.05.2017.

[27] MobiVoc Website, Schema. Mobivoc: Open mobility vocabulary. webpage, 2017. `http://schema.mobivoc.org/`. Accessed 02.05.2017.

[28] Mulligan, C. E. A., and Olsson, M. Architectural implications of smart city business models: An evolutionary perspective. *IEEE Communication Magazine 51*, 6 (2013), 80 – 85.

[29] Nixon, L., Simperl, E., Krummenacher, R., and Martin-recuerda, F. Tuplespace-Based Computing for the Semantic Web: A Survey of the State-of-the-Art. *Knowl. Eng. Rev. 23* (2008), 181–121.

[30] OASIS AMQP Technical Committee. Oasis amqp version 1.0, sections 2.6.12-2.6.13. Accessed 10.05.2017.

[31] Pike Research. On smart cities. webpage, 2017. `http://smartcitiescouncil.com/tags/pike-research`. Accessed 02.09.2017.

[32] Prophet.com. Interoperability: The challenge facing the internet of things. webpage, 2014. `https://www.prophet.com/thinking/2014/02/interoperability-the-challenge-facing-the-internet-of-things/`. Accessed 02.05.2017.

[33] Saint-Andre, P. Extensible messaging and presence protocol (xmpp): Core, 2011. https://tools.ietf.org/html/rfc6120. Accessed 11.05.2017.

[34] Salman, T. Internet of things protocols and standards. webpage, 2015. http://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_prot/index.html. Accessed 04.05.2017.

[35] Schaffers, H., Komninos, N., Pallot, M., Trousse, B., Nilsson, M., and Oliveira, A. Smart cities and the future internet: Towards cooperation frameworks for open innovation. *The Future Internet, Lecture Notes Computer Science 6656* (2011), 431 – 446.

[36] Schema.org website. Welcome to schema.org. webpage, 2017. http://http://schema.org/.. Accessed 02.05.2017.

[37] Schneider, S. Understanding the protocols behind the internet of things. *Electronic Design* (2013). http://www.electronicdesign.com/iot/understanding-protocols-behind-internet-things. Accessed 04.05.2017.

[38] Shelby, Z., Hartke, K., and Bormann, C. The constrained application protocol (coap). *RFC 7252* (2014).

[39] Tai, S., and Rouvellou, I. Strategies for integrating messaging and distributed object transactions. *In IFIP/ACM International Conference on Distributed systems platform* (2000), 308–330.

[40] The bIoTope website. The biotope project. webpage, 2017. http://www.biotope-project.eu/. Accessed 17.09.2017.

[41] The MQTT protocol official website. Interoperability: The challenge facing the internet of things. webpage, 2014. http://www.mqtt.org.. Accessed 02.05.2017.

[42] The Open Group. Open data format (o-df), an open group internet of things (iot) standard, 2014. http://www.opengroup.org/iot/odf/. Accessed 10.05.2017.

[43] The Open Group. Open messaging interface (o-mi), an open group internet of things (iot) standard, 2014. http://www.opengroup.org/iot/omi/index.htm. Accessed 10.05.2017.

[44] Tracy, P. Iot interoperability: Where it stands and what comes next. webpage, 2016. `http://www.rcrwireless.com/20161031/internet-of-things/iot-interoperability-tag31-tag99`. Accessed 02.05.2017.

[45] Treese, W. The open market internet index for 11 november 1995. webpage, 1995. `Treese.org`. Accessed 02.09.2017.

[46] University of Bonn Website. About project mobivoc. webpage, 2017. `http://eis.iai.uni-bonn.de/Projects/MobiVoc.html`. Accessed 02.05.2017.

[47] Weiser, M. The Computer for the Twenty-First Century. *Scientific American 265*, 3 (1991), 94–104.

Appendix A

Parking Service O-DF examples

In the appendix A there is O-DF structure of the parking service. In these three figures there are O-DF structures of the parking service element (Figure A.1), parking space types (Figure A.2) and parking spot (Figure A.3).

```
<omiEnvelope ttl="10" version="1.0" xmlns="http://www.opengroup.org/xsd/omi/1.0/">
  <response>
    <result msgformat="odf">
      <return returnCode="200">
      </return>
      <msg>
        <Objects xmlns="http://www.opengroup.org/xsd/odf/1.0/" xmlns:xs="http://
www.w3.org/2001/XMLSchema" xmlns:odf="http://www.opengroup.org/xsd/odf/1.0/">
          <Object>
            <id>ParkingService</id>
            <Object type="list">
              <id>ParkingFacilities</id>
              <Object type="schema:ParkingFacility">
                <id>CSBuildingParkingLot</id>
                <InfoItem name="MaxParkingHours">
                  <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">20</value>
                </InfoItem>
                <InfoItem type="mv:isOwnedBy" name="Owner">
                  <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Aalto University</value>
                </InfoItem>
                <Object type="schema:OpeningHoursSpecification">
                  <id>openingHoursSpecification</id>
                  <InfoItem name="opens">
                    <value unixTime="1496236429" type="schema:Time"
dateTime="2017-05-31T16:13:49.394+03:00">00:00</value>
                  </InfoItem>
                  <InfoItem name="closes">
                    <value unixTime="1496236429" type="schema:Time"
dateTime="2017-05-31T16:13:49.394+03:00">24:00</value>
                  </InfoItem>
                </Object>
                <Object type="schema:GeoCoordinates">
                  <id>geo</id>
                  <InfoItem name="latitude">
                    <value unixTime="1496236429" type="xs:double"
dateTime="2017-05-31T16:13:49.394+03:00">60.187556</value>
                  </InfoItem>
                  <InfoItem name="longitude">
                    <value unixTime="1496236429" type="xs:double"
dateTime="2017-05-31T16:13:49.394+03:00">24.8213216</value>
                  </InfoItem>
                  <Object type="schema:PostalAddress">
                    <id>address</id>
                    <InfoItem name="addressCountry">
                      <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Finland</value>
                    </InfoItem>
                    <InfoItem name="streetAddress">
                      <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Konemiehentie 4</value>
                    </InfoItem>
                    <InfoItem name="addressRegion">
                      <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Espoo</value>
                    </InfoItem>
                    <InfoItem name="addressLocality">
                      <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Otaniemi</value>
                    </InfoItem>
                    <InfoItem name="postalCode">
                      <value unixTime="1496236429" type="xs:double"
dateTime="2017-05-31T16:13:49.394+03:00">2150.0</value>
                    </InfoItem>
                  </Object>
                </Object>
```

Figure A.1: Parking Service O-DF

```
<Object type="list">
                  <id>ParkingSpaceTypes</id>
                  <Object type="mv:ParkingUsageType">
                    <id>ElectricVehicleParkingSpace</id>
                    <InfoItem type="mv:hasVehicleLengthLimitInM"
name="MaxLength">
                      <value unixTime="1496236429" type="xs:float"
dateTime="2017-05-31T16:13:49.394+03:00">2.5</value>
                    </InfoItem>
                    <InfoItem type="mv:hasTotalCapacity" name="TotalCapacity">
                      <value unixTime="1496236429" type="xs:int"
dateTime="2017-05-31T16:13:49.394+03:00">2</value>
                    </InfoItem>
                    <InfoItem type="mv:hasVehicleWidthLimitInM" name="MaxWidth">
                      <value unixTime="1496236429" type="xs:float"
dateTime="2017-05-31T16:13:49.394+03:00">2.5</value>
                    </InfoItem>
                    <InfoItem type="mv:hasNumberOfVacantParkingSpaces"
name="NumberOfVacantParkingSpaces">
                      <value unixTime="1496236429" type="xs:int"
dateTime="2017-05-31T16:13:49.394+03:00">2</value>
                    </InfoItem>
                    <InfoItem type="mv:PriceParking" name="HourlyPrice">
                      <value unixTime="1496236429" type="xs:int"
dateTime="2017-05-31T16:13:49.394+03:00">2</value>
                    </InfoItem>
                    <InfoItem type="mv:hasVehicleHeightLimitInM"
name="MaxHeight">
                      <value unixTime="1496236429" type="xs:float"
dateTime="2017-05-31T16:13:49.394+03:00">2.5</value>
                    </InfoItem>
```

Figure A.2: Parking Space Types O-DF

```
<Object type="list">
                      <id>Spaces</id>
                      <Object type="mv:ParkingSpace">
                        <id>EVSpace2</id>
                        <InfoItem name="Available">
                          <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">true</value>
                        </InfoItem>
                        <InfoItem name="User">
                          <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">NONE</value>
                        </InfoItem>
                        <Object type="mv:Charger">
                          <id>Charger</id>
                          <InfoItem type="mv:Model" name="Model">
                            <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">ExampleI</value>
                          </InfoItem>
                          <InfoItem type="mv:Brand" name="Brand">
                            <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Exampler</value>
                          </InfoItem>
                          <Object type="mv:Plug">
                            <id>Plug</id>
                            <InfoItem type="mv:CableAvailable"
name="CableAvailable">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">false</value>
                            </InfoItem>
                            <InfoItem type="mv:ChargingSpeed"
name="ChargingSpeed">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">mv:Standard</value>
                            </InfoItem>
                            <InfoItem type="mv:PlugType" name="PlugType">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">Schuko Plug</value>
                            </InfoItem>
                            <InfoItem type="mv:Voltage" name="Voltage">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">240V</value>
                            </InfoItem>
                            <InfoItem type="mv:Power" name="Power">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">2400W</value>
                            </InfoItem>
                            <InfoItem type="mv:LockerAvailable"
name="LockerAvailable">
                              <value unixTime="1496236429"
dateTime="2017-05-31T16:13:49.394+03:00">true</value>
                            </InfoItem>
                          </Object>
                        </Object>
                      </Object>
```

Figure A.3: Parking Spot O-DF

Appendix B

O-DF elements mapped to POJO

O-DF Object Element	POJO
`ParkingService`	ParkingService.java
`<Object type="list">` `<id>ParkingFacilities</id>`	ParkingLot.java

`<Object`
`type="schema:OpeningHoursSpecification">`
`<id>openingHoursSpecification</id>`
	OpeningHours.java

`<Object`
`type="schema:GeoCoordinates">`
`<id>geo</id>`
	GeoCoordinates.java

`<Object`
`type="schema:PostalAddress">`
`<id>address</id>`
	Address.java

`<id>ParkingSpaceTypes</id>`
`<Object`
`type="mv:ParkingUsageType">`
	ParkingSection.java

`<id>Spaces</id>` `<Object type="mv:ParkingSpace">`	ParkingSpot.java
`<Object type="mv:Charger">` `<id>Charger</id>`	Charger.java
`<Object type="mv:Plug">` `<id>Plug</id>`	Plug.java

Figure B.1: O-DF elements to POJO

Appendix C

O-DF queries example

In Appendix C there are collected the examples of queries to the server for different operations: finding the parking lots of desired location (Figure C.1), reservation of parking spot (Figure C.2), unbooking (leaving) the parking spot (Figure C.3), opening the lid of the EV charger (Figure C.4).

```
<omiEnvelope xmlns="http://www.opengroup.org/xsd/omi/1.0/" version="1.0"
ttl="0">
  <call msgformat="odf">
    <msg>
      <Objects xmlns="http://www.opengroup.org/xsd/odf/1.0/">
        <Object>
          <id>ParkingService</id>
          <InfoItem name="FindParking">
            <value type="odf">
              <Objects>
                  <!-- Contains only required parameters -->
                      <Object type="FindParkingRequest">
                        <id>Parameters</id>
                        <description lang="English">List of possible
parameters to request.</description>
                        <InfoItem name="ParkingUsageType">
                          <value unixTime="1495635263"
dateTime="2017-05-24T17:14:23.724+03:00">mv:ElectricVehicleParkingSpace</
value><!-- Works without mv: too -->
                        </InfoItem>
                        <Object type="schema:GeoCoordinates">
                          <id>Destination</id>
                          <InfoItem name="latitude">
                            <value unixTime="1495635263" type="xs:double"
dateTime="2017-05-24T17:14:23.724+03:00">60.187556</value>
                          </InfoItem>
                          <InfoItem name="longitude">
                            <value unixTime="1495635263" type="xs:double"
dateTime="2017-05-24T17:14:23.724+03:00">24.8213216</value>
                          </InfoItem>
                        </Object>
                      </Object>
                    </Objects>
            </value>
          </InfoItem>
        </Object>
      </Objects>
    </msg>
  </call>
</omiEnvelope>
```

Figure C.1: Find Parking Request

```
<omiEnvelope xmlns="http://www.opengroup.org/xsd/omi/1.0/" version="1.0"
ttl="0">
  <write msgformat="odf">
    <msg>
      <Objects xmlns="http://www.opengroup.org/xsd/odf/1.0/">
        <Object>
          <id>ParkingService</id>
          <Object>
            <id>ParkingFacilities</id>
            <Object>
              <id>CSBuildingParkingLot</id>
              <Object>
                <id>ParkingSpaceTypes</id>
                <Object>
                  <id>ElectricVehicleParkingSpace</id>
                  <Object>
                    <id>Spaces</id>
                    <Object type="mv:ParkingSpace"><!-- TYPE ATTRIBUTE
MUST BE GIVEN AGENT USES IT FOR PARSING! -->
                      <id>EVSpace1</id>
                      <InfoItem name="Available">
                        <value>false</value>
                      </InfoItem>
                      <InfoItem name="User">
                        <value>YOUR_USERNAME</value>
                      </InfoItem>
                      <id>Charger</id>
                          <InfoItem name="LidStatus">
                              <value>Open</value>
                          </InfoItem>
                      </Object>
                    </Object>
                  </Object>
                </Object>
              </Object>
            </Object>
          </Object>
        </Object>
      </Objects>
    </msg>
  </write>
</omiEnvelope>
```

Figure C.2: Reservation of Parking Spot Request

```
<omiEnvelope xmlns="http://www.opengroup.org/xsd/omi/1.0/" version="1.0"
ttl="0">
  <write msgformat="odf">
    <msg>
      <Objects xmlns="http://www.opengroup.org/xsd/odf/1.0/">
        <Object>
          <id>ParkingService</id>
          <Object>
            <id>ParkingFacilities</id>
            <Object>
              <id>CSBuildingParkingLot</id>
              <Object>
                <id>ParkingSpaceTypes</id>
                <Object>
                  <id>ElectricVehicleParkingSpace</id>
                  <Object>
                    <id>Spaces</id>
                    <Object type="mv:ParkingSpace"><!-- TYPE ATTRIBUTE
MUST BE GIVEN AGENT USES IT FOR PARSING! -->
                      <id>EVSpace1</id>
                      <InfoItem name="Available">
                        <value>true</value>
                      </InfoItem>
                      <InfoItem name="User">
                        <value>YOUR_USERNAME</value><!--Service will
check that this is same as the current user, but will actually write NONE
to it. -->
                      </InfoItem>
                      <Object><!-- If not given when freeing space. Lid
cannot be opened afterwards without creating new reservation.-->
                          <id>Charger</id>
                          <InfoItem name="LidStatus">
                              <value>Open</value>
                          </InfoItem>
                      </Object>
                    </Object>
                  </Object>
                </Object>
              </Object>
            </Object>
          </Object>
        </Object>
      </Objects>
    </msg>
  </write>
</omiEnvelope>
```

Figure C.3: Unbooking of Parking Spot Request

```
<omiEnvelope xmlns="http://www.opengroup.org/xsd/omi/1.0/" version="1.0"
ttl="0">
  <write msgformat="odf">
    <msg>
      <Objects xmlns="http://www.opengroup.org/xsd/odf/1.0/">
        <Object>
          <id>ParkingService</id>
          <Object>
            <id>ParkingFacilities</id>
            <Object>
              <id>CSBuildingParkingLot</id>
              <Object>
                <id>ParkingSpaceTypes</id>
                <Object>
                  <id>ElectricVehicleParkingSpace</id>
                  <Object>
                    <id>Spaces</id>
                    <Object type="mv:ParkingSpace"><!-- TYPE ATTRIBUTE
MUST BE GIVEN AGENT USES IT FOR PARSING! -->
                      <id>EVSpace1</id>
                      <InfoItem name="User">
                        <value>YOUR_USERNAME</value><!-- Service will
check that this is same as current user's username before opening the
lid. -->
                      </InfoItem>
                      <Object>
                          <id>Charger</id>
                          <InfoItem name="LidStatus">
                              <value>Open</value>
                          </InfoItem>
                      </Object>
                    </Object>
                  </Object>
                </Object>
              </Object>
            </Object>
          </Object>
        </Object>
      </Objects>
    </msg>
  </write>
</omiEnvelope>
```

Figure C.4: Open the Charger Lid Request