JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Ramez Elbaroudy**

Submitted at
**Research Institute for Symbolic Computation**

Supervisor
**A. Univ.-Prof. DI Dr. Wolfgang Schreiner**

07 2018

# A Gateway for the Generic Conversion of Protocols for Smart Meters and IoT Applications

Master Thesis

to obtain the academic degree of

Master of Science

in the Master's Program

Internationaler Universitätslehrgang: Informatics: Engineering & Management

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

# Acknowledgment

In the beginning, I would like to thank Prof. Dr.Dr.h.c.mult. Bruno Buchberger for accepting me in the ISI program and giving me the opportunity to have my masters degree from Johannes Kepler University Linz. I would like to thank Prof. Dr. Slim Abdennadher (the Study Dean of Engineering Faculties in German University in Cairo) for recommending this program for me, and for being motivational and influential character through out my study at the German University in Cairo.

I would like to express my appreciation and gratitude to my university adviser A. Univ.-Prof. DI Dr. Wolfgang Schreiner for all the academic efforts and the extremely useful advises in all parts of the project. I would like to thank Prof. Schreiner for his guidance, patients and assistance in my master's project.

I would like also to acknowledge the help of S&T supervisors Mr. Pumberger Ludwig and Mr. Gaderbauer Helmut, for the constant help, not only providing technical help throughout the different phases of the project, but also for providing the hardware for the project, namely smart meters and a laptop.

Our stay in Linz would not have been extremely easy without the help of Mrs. Betina Curtis, starting from the day of acceptance of the master's program till the day of graduation. I am grateful to her careful coordination and organizational skills.

I would like to thank my parents for allowing me to have such an opportunity with non stop encouragement and support without which this work would have been impossible.

Last but not least, I would like to thank my dear ISI colleges and friends for the fantastic time we spent together, and for making our daily life such unique experience, thank you for the amazing company.

# Abstract

In the recent years the number of Internet of things (IoT) devices have increased in a remarkable way. Due to this increase of devices many challenges have appeared. One of the most markable challenge is interoperability challenge, where devices use different standards, protocols and conventions to exchange information. This challenge has appeared because of the different devices' specifications and the different manufacturers of the different types of IoT devices. The current thesis proposes a generic API that can be used for communication with different IoT devices, which use different conventions and protocols; understanding the underlying conventions or protocols used. In this thesis, we used smart meters as the IoT devices for testing the generic API. The generic API is first described in an abstract way; then we describe the Java binding in order to use the API with the Java programming language. The implemented generic API enables developers to communicate with different IoT devices without the need of knowing the specification of each protocol. The current thesis can be considered as an important point of extending the research field of interoperability of IoT devices. Furthermore, the generic API can be extended in order to support other types of protocols and conventions.

# Contents

# List of Figures

# Chapter 1

# Introduction

After Internet and mobile communication, the Internet of Things (IoT) is regarded as the third wave of information technology. The Internet of Things allows objects to be sensed or controlled remotely across existing network infrastructures, creating opportunities for more direct integration of the physical world into computer-based systems, and resulting in improved efficiency, accuracy and economic benefits in addition to reduced human intervention; therefore it saves money and time. When IoT is augmented with sensors and actuators, the technology becomes an instance of the more general class of cyber-physical systems, which also encompasses technologies such as smart grids, virtual power plants, smart homes, intelligent transportation and smart cities. A lot of objects and devices are exploited by IoT, such as cars, televisions, bridges, clothes, smart meters, streets, and cities in order to improve the quality of life of people using these objects by allowing automation. With the exponential growth of the IoT objects connectivity, many challenges were introduced to the IoT field, such as security reliability and fault tolerance. One of the challenges that was introduced to the IoT field is interoperability. With the increase of the heterogeneous objects that depends on each other, also the connectivity challenges within the IoT system increased. In addition, IoT devices can collect data in different forms, which brings different interoperability challenges to IoT system. Also legacy systems such as legacy protocols add to the interoperability challenge in an IoT system [14].

With the increased cost of energy as well as the concern of global warming the concept of smart grids has been introduced. A smart grid is an electric grid that includes several energy measures such as smart meters and renewable energy resources. The advantages of using a smart grid is to reduce the CO2 emissions as well as to reduce the energy cost for the customer and to ensure sustainability [11][28].

A smart meter is an electronic device that measures the consumption of electricity, gas, and water. In general, a smart meter can measure the consumption of combined set of utilities, such as electricity and gas combined, which saves money, as the owner of the property does not need to buy several smart meters to measure each utility. Smart meters help in the conservation of energy as they provide more detailed information compared to classical meters. The energy consumption is communicated to both the utility service and the customer through a website, in home displays, and mobile applications [36]. This means that the owner of a smart meter can track down his consumption, as a motivation for more energy saving. In addition, smart meters allow bidirectional communication;

therefore not only can the data collection service or the cloud provider can not only receive data from smart meter, but it can also send data or information to the smart meter. For example, when an owner leaves the house and new a customer rents the same house, the data collection service can reset the power consumption measure. Nowadays, smart meters have been deployed in big numbers around the world [10]. Smart meters have different manufacturers; based on the manufacturer, the protocol used by the smart meter can be different. Therefore, gateways need to be introduced in order to ease the integration of the different sensors and convert the different protocols of different devices in a smart city to an end protocol that will be used for communication with the cloud or service provider [38].

The gateway (also known as a data concentrator) acts as a middle man that supports bi-directional communication; for example, a utility service could order a specific smart meter to shut down, as the owner of the smart meter is not currently staying in the house, for example, if he went for a vacation. In the other direction, the meter sends the energy consumed by the customer to the utility provider. A gateway should be able to collect data from different meters encrypted and send it to the cloud provider without decrypting it. The information of several meters should be sent at once to the provider. According to [15], in Germany the central communication components in the future smart metering infrastructure will be such gateways.

The main goal of this thesis is to overcome the interoperability challenges by designing a generic API that will allow developers in the cloud provider side or data collection service to query the smart meter for data or send data to the smart meter, without knowing the underlying protocol specification used by the smart meter. There are many components in the smart meter technology specifically and in IoT application in general that can cause the interoperability issues. In this thesis, we overcome the different types of challenges related to incompatibility of smart meters connection with the data collection service. One of the main causes of the interoperability issues, is that smart meters use different protocols and there does not exist a unified protocol that is used by the smart meter technology.

To approach the goal of the thesis, we first examined and investigate different protocols used by smart meters. In this thesis we were using two different protocols smart meters, namely DLMS/COSEM and ANSI C12.18 smart meters. We examined the application layer in both protocols and compare their features in details. Then, after collecting the information regarding both protocols, we designed a generic API. Which covers the features of both the DLMS/COSEM and the ANSI C12.18 protocol. In order to implement the generic API we designed several objects, data structures and algorithms in order to overcome the interoperability issue. In particlare, we designed a data structure, namely dataStore that is used in the implementation of the generic API to overcomes the data type issue that occurs within smart meters.

The remainder of this thesis is as follows. In Chapter 2, we discuss in detail the related work with an explanation of the context of the problem in details; in particular we investigate some of the solutions proposed either in research papers or in the industry to solve the interoperability issues of IoT devices. In Chapter 3, we investigate both types of protocols used in the project and explain the application layer as well as the features of each protocol. In chapter 4, we describe the suggested generic API in abstract way, by the building blocks of the API, without specifying either the implementation or the binding to a programing language. In Chapter 5, we describe the binding of the generic API to the programming languages, Java. Finally in Chapter 6 we give a summary and suggestions for future work.

Chapter 2

# Related Work

This chapter presents an overview of the basic concepts covered in this thesis. At the beginning, we are going to discuss the context of the problem. Then, the chapter gives an overview of the projects related to this work whether in industry or in research.

## 2.1 Context of the Problem

Nowadays the number of IoT devices have been increasing tremendously, this increase will only keep accelerating. Smart meters are IoT devices that measure energy consumption at any part of time and are capable of sending recorded information to other devices. Smart meters are one of the key components in smart city concept. According to [4] by 2020 the European union aims to change a minimum of 80% of electricity meters with smart meters, as this will reduce energy bills, which saves money for consumers [4]. In 2014 the European commission reported that around 200 million smart meters for electricity and gas will be disturbed by 2020 in the European union which has potential investment of €45 billion [4]. In this section we will discuss the differences between smart meters and regular meters, highlighting the advantages of using smart meters; then we will discuss the challenges that face the smart meters industry.

### 2.1.1 Smart Meters Advantages

One of the key difference between smart meters and regular meters is that smart meters are capable of sending the information recorded by the meter at any time to other devices [11]. Therefore, the energy consumer can have the data related to his consumption using tracking device, such as mobile devices. This feature allows user to monitor frequently his energy consumption, thus the meter creates self awareness about the consumption and can help in reducing it. Smart meters measure the energy consumption more often than regular meters and send reports of energy consumption more frequently, at least once a day. Another key feature of smart meters is the two way communication channel between a smart meter and a data collection service, which allows the data collection service to send information to the smart meter [11]. One application where this feature can be useful is that when a person leaves the house for vacation, the data collection service can turn off the meter till the person comes back to the house. In general, smart meters can measure electricity, water, or gas depending on the manufacturer of the smart meter. The key advantages of using smart meters over regular meter are highlighted below [19]:

- Smart meters allow consumers to access the data for managing their energy usage.
- Smart meters measure highly accurate billings compared to regular meters.
- Smart meters offer effective outage repair.
- Smart meters allow different rates tariff options for the consumer.
- For service provider, smart meters eliminate manual reading, and its expenses.
- Data collection services will have access to more data for further studies.
- Utility providers have the ability to remotely connect and disconnect the smart meters.
- Smart meter systems support further programs of large smart grid and smart city initiatives, which will have environmental benefits.

### 2.1.2 Challenges in the Smart Meters Industry

With such an increase of smart meters, whether for gas or for electricity, several challenges have appeared for the smart meter's manufacturers and data collection services which include:

- Security concerns.
- Fault tolerance concerns.
- Privacy concerns.
- Interoperability concerns.
- Data integrity concerns.

All these challenges add requirements for the manufactures of smart meters to ensure security and privacy of meters by encryption, authentication and privacy of meter data. In this thesis we are going to discuss and provide a solution for the interoperability challenge that faces smart meters [19].

### 2.1.3 Interoperability Challenge

As discussed in the previous subsection, there are several challenges that face the smart meters industry. In this thesis, we are going to discuss the interoperability challenge in detail and we will provide a system that is capable of overcoming this challenge. Interoperability is a challenge that not only faces the smart meter industry, but it also faces IoT applications in general.

As mentioned in the previous sections, there has been an increase of smart meters production. With this increase, a large number of manufacturers has been trying to manufacture smart meters. Each manufacturer has his own specification of smart meters. The diversity of the manufacturers introduced the interoperability challenge as meters may vary in different parameters, which cause inconsistency in the technologies used. In addition, this inconsistency makes it difficult for data collection services to communicate with different types of smart meters, which may lead developers to invest a massive amount of time to program specific solutions for each type of smart meter, which adds to the cost of the data collection service. In this thesis, we are going to introduce a system that uses generic API, which allows a data collection service to communicate with any type of smart meter, where the implementation of the API solves the underlaying low level interoperability issues. In the next subsections, we are going to highlight the different parameters that can cause interoperability issues between different types of smart meters.

### 2.1.4 Protocols Diversity

One of the most important issues that can cause incompatibility is the protocol used used by smart meters; there is a wide variety of protocols and it depends on the manufacturer which one to use. This variety of protocols add challenges as follows:

- Different parameters in initiating connection.
- Different methods of data transfer.
- Different parameters in closing a connection.

There are several protocols specific to the smart meters domain. DLMS/COSEM is a popular protocol for exchanging metering data [17]. DLMS stands for Device Language Message Specification which offers an abstract concept for communication, while COSEM stands for Companion Specification for Energy Metering which represents a set of rules for data exchange with metering equipment [9]. Another protocol popular in north America is ANSI C12.18 [18], which is an ANSI standard for describing the two-way communication with metering equipments. Open Smart Grid Protocol [27] is another protocol for metering data exchange, which was issued by the European Telecommunications Standards Institute; however for the Open Smart Grid Protocol several security flaws were identified [20] [21]. Recently, there has been a growing trend of using more general protocol not specific to the metering domain, such as

TCP/IP which will allows an universal metering interface for the production of smart meters as well as smart grid devices. In this thesis our proposed implementation of generic API is going to support the two most popular protocols DLMS/COSEM and ANSI C12.18.

#### 2.1.4.1 Naming Conventions Diversity

Another challenge that can face developers in terms of interoperability is difference in naming conventions, which may cause difficulties reading information from registers in a smart meter. When reading data from a smart meter, the data collection service should indicate which register to read from. Register names vary, not only from protocol to protocol, but it also from manufacturer to manufacturer. For example, the DLMS/COSEM protocol allows the usage of two naming conventions, namely short naming and logic naming. Some manufacturers allow only the short naming convention, while other manufacturers allow only the logic naming convention. Also some manufacturers can allow both types of naming convention giving the developer freedom to use whatever naming convention he prefers. Not to mention, some protocols may use naming conventions totally different from short naming and logic naming.

However, in the recent years a standardized naming convention has been introduced under the name Object Identification System (OBIS). OBIS offers standard identifiers for all data within smart meters. Data can be either measurement values or abstract values, usually stored inside registers which are inside the smart meter. OBIS consist of six value groups in a hierarchal structure. The groups are counted from group A to group F which are divided as follows [3]:

- **Group A:** Determines energy type read.
- **Group B:** Determines the channels to be measured.
- **Group C:** Determines the measured physical quantity.
- **Group D:** Determines the handling methods and codes related to certain countries.
- **Group E:** Determine tariff rates.
- **Group F:** Determine values recorded in the past.

Groups B, C and D have code space which can allow for manufacturers to assign their own identifiers, leaving more freedom for manufacturers. In modern DLMS/COSEM smart meters, every COSEM object is distinguished by the logical name given by OBIS, class identifier and version. However in old DLMS/-COSEM smart meters it can use different form of naming convention depending on the manufacturer of the smart meter [2].

#### 2.1.4.2 Data Types Diversity

Different data types of measured and abstract values in the smart meter add challenges for developers when accessing data. For example, a smart meter's register that measures positive active energy could return a double value while in another smart meter that use different protocol can return an integer value; this inconsistency could result in the loss of information when casting double to integer; moreover, it can result in incompatibility errors. One of the causes of such differences in data types is the inconsistency in units which change the implementation of numbers in the data types. The interface class definition of OBIS offers a standard data type of each data value in the metering equipment [2]. Therefore, OBIS allows the standardization of the data values across metering equipments whether measure values or abstract values.

#### 2.1.4.3 Units Diversity

The final issue that cause inconsistency through different metering equipments is units diversity. This issue occurs when a smart meter measures the reading value of a certain register in a unit, while another meter measures the reading value of a certain register in another unit. For example, a smart meter may measure positive active energy in kWh, while another smart meter may measure positive active energy in Wh; this causes not only inconsistency errors in terms of units, but it also causes issues such as, that two equal readings may have different values, which opens the room for confusion.

### 2.1.5 S&T

The thesis project is developed in cooperation with the S&T company [30]. In this subsection, we are going to discuss the profile of S&T and how our project helps the company with the challenges it face.

S&T was founded in 2008; today it is part of the Exchange's TecDAX index of leading high-techs. The company has some 3,700 staff members working for the Group's subsidiaries and operations, which are located in more than 25 countries. S&T's clients range from SMEs that are active in the widest variety of sectors to leading groups that operate on a world-spanning basis [1]. S&T has several technology departments including automation solution, enterprise security, software development and smart energy. This project is developed in cooperation with the smart energy department were the project is part of end-to-end facilitation of smart-energy projects, smart grids and smart metering solutions.

S&T has different customers using large amount of different IoT devices, each have its own protocol, which causes interoperability issues; therefore this adds challenges to the development of systems for each protocol used by every IoT device, which adds to the development cost. Our suggested system offers a generic API that is capable of connecting with different protocols without knowing the specific protocols that the different IoT devices use. In addition S&T has customers that have smart meters which use different protocols and have different manufactures. In this project we decided to focus on smart meters as the IoT devices, because not only do smart meters have interoperability issues, such as different protocols and manufactures, but also smart metes are handy and much easier to connect to a gateway for experiments compared to other IoT devices. S&T helped us with the technology supplies: two smart meters that used different protocols (DLMS/COSEM and ANSI C12.18) was handed out to us for the development and testing of the generic API.

## 2.2 State of the Art

### 2.2.1 In Research

One of the biggest challenges that face the IoT field is the interoperability challenge, due to the different protocols used by the devices. The research described in [38] was able to design a gateway that was developed in order to convert protocols of different sensor network protocols. This gateway is based on the protocols Zigbee and GPRS to handle telecommunication requirements and IoT application scenarios in general. Figure 2.1 [38] shows a scenario of using this gateway in a smart home. A smart home represent a house that contains different types of sensors that measure set of data constantly. These sensors are used in order to help providing a house that is comfortable as well as environmental. The "on field" gateway plays an extremely crucial role in interconnecting the different IoT devices together, although each device uses is own protocol and specification.

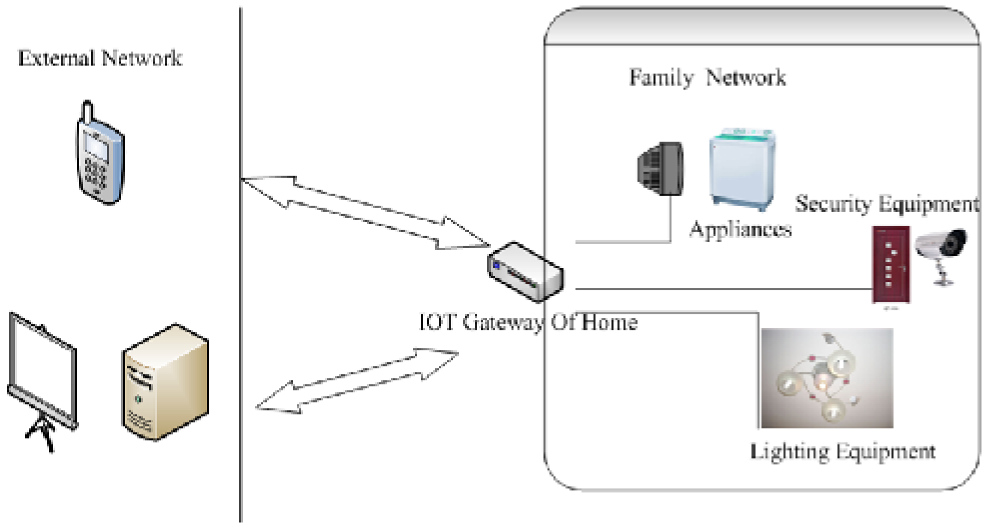


Figure 2.1: Scenario of Using the Gateway in Smart Home (from [38])

The designed gateway software architecture is composed of three different modules, shown in Figure 2.2 and described as follows:

- **Sensor Node Module:** A sensor node module represents the perception layer; it collects the sensed data and send the collected data to the gateway. The sensor node module also can receive commands from the gateway.

- **Gateway Module:** Represents the bridge between the sensor node and the application layer: after receiving data from this module sensor node, it transmits the data to the application layer. The designed system has a Ethernet interaction module and a GPRS interaction module to enable communication with the application layer. It also has a protocol conversion module to enable the protocol conversion to overcome the interoperability issue discussed before.

- **Management Platform:** The management layer contains an application platform, which communicates with the gateway in order to manage the gateway and the sensor network. The gateway provides a user interface to allow the control of sensor nodes through the gateway.

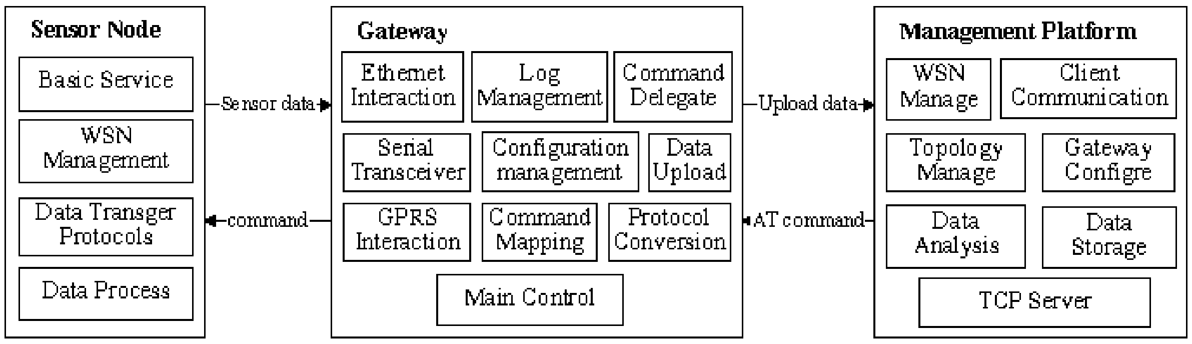


Figure 2.2: Software Architecture of the Gateway (from [15])

Gateways act as a middle man between data collection service or utility provider and the IoT appliance. According to [15], gateways connected to smart meters have two main functionalities: First, they collect data from all different smart meters connected to it. The second functionality is that gateways offer an interface in order to enable easier data exchange with the smart meters. Gateways retrieve data from the smart meters according to different parameters as follows [15]:

- **Meter Profiles:** Meter profilers indicate the configuration of the smart meter for energy providing and measuring consumption.

- **Time-stamps:** Each measured value or data exchange between a smart meter and the gateway has a time-stamp which indicates the exact time when the last measurement happened or the last data exchange happened.

- **Tariffing Information:** One of the biggest advantages of smart meters is having different tariffs based on the time of the day or the energy load from the city point of view. Whenever a new value is read in from the smart meter, such as power consumption, the value related to the tariff at that point of time should also be read.

"Ambient Assisted Living" [37] is improving the health care as well as the prognosis of different types of diseases by artificial intelligent and Internet of Things, by monitoring the health status of the patient continuously and learn behaviors and patterns in order to provide more accurate diagnosis. According to [37], interoperability, system security, dynamic increase in storage, and streaming quality of service are challenges that face IoT and multimedia technologies in the field of health care. Health care uses IoT devices and sensors for applications such as ambient aiding living and telemedicine; therefore [37] suggested a system that is able to overcome these challenges for an ambient assistant living system. We are going to discuss how the system overcame the interoperability issue.

The medical and health field faces interoperability challenges especially in telemedicine and ambient assisted living, because an ambient assisted living system needs to be connected to different devices that sense different objects, such as blood pressure and blood-glucose using different meters. Ambient assisted living supports wireless and wired connection based on protocols that are used by most devices, such as the Bluetooth Health Device Profile (HDP) and the ZigBee Health CareTM Profile. However, for wired connection the USB Personal Healthcare Device Class (PHDC) is th most used, as it is recommended by CHA. Meters have different manufacturers; therefore each meter has its own specification. The Continua Health Alliance (CHA) is international, non-profit, authorized telemedicine and telehealth standard body. The suggested IoT gateway follows the CHA specification which suggests two layers for overcoming the interoperability issues in IoT health gateway. The first layer is the transplantation layer which is responsible for a wireless and wired communication health standard. In the data layer the frequent basic framework protocol represent optimized exchanging model and application profile. In addition, the framework protocol defines message types, data types and communication models. The designed system was implemented on hardware gateway as Windows XP desktop PC. Figure 2.3 shows the architecture developed by [37] based on the CHA framework; the designed system is able to overcome the challenges of interoperability, system security, streaming quality of service, and dynamic increase in storage in the field of ambient aiding living and telemedicine [37].

After the data have been collected by the gateway, these are stored in the database for the utility provider or the data collection service. In general gateways can have one of two forms, either hardware-gateway or software-gateway. A software-gateway was developed by [15] to gather data from different smart meters. The gateway was developed as a Java-based smart meter gateway framework in order to gather the data and send them to external market participants such as utility providers and data collection services. The developed gateway has a Graphical user interface (GUI) for configuration, such as connecting more meters to the gateway.

### 2.2.2 In Industry

In the industry there exist several vendors for IoT gateways, such as Microsoft Azure [25], IBM Watson [16], AWS [7] and Google cloud service [12]. However, all of the vendors, except Microsoft Azure, do not produce gateway that enables protocol conversion. The Microsoft Azure IoT protocol gateway is the most popular protocol gateway in industry. Other vendors' gateways offers different capabilities. For example, the IBM Watson software for gateways enables devices to work as a gateway by introducing

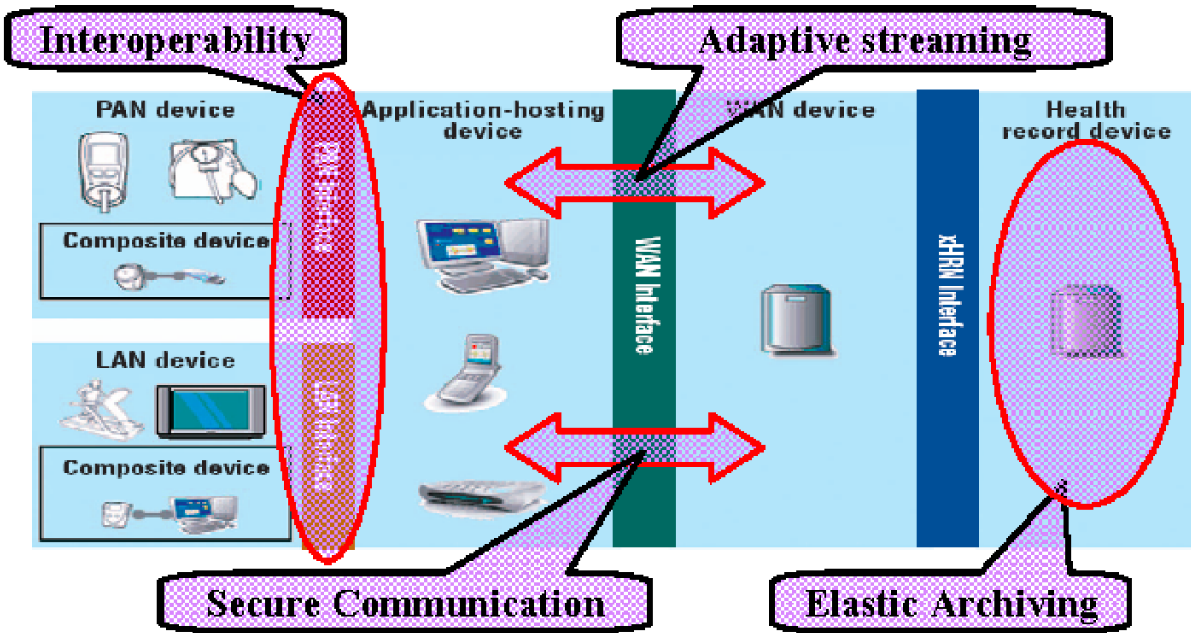


Figure 2.3: Architecture of the Ambient Assisted Living System (from [37])

functionalities to communicate with connecting devices such as single connection, device management, and automatic registration [35]. However, the IBM Watson gateway does not handle the protocol interoperability issues.

In the following section, we are going to discuss the product Microsoft Azure IoT protocol in more detail as it is the most popular gateway software product that supports protocol conversion.

#### 2.2.2.1 Azure IoT Protocol Gateway

The Azure IoT protocol gateway is a getaway designed by Microsoft, that supports protocol adaptation and conversion framework. The sole purpose of the Azure IoT protocol gateway is that the gateway overcomes the interoperability issue, unlike many other gateways that were previously discussed. The Azure IoT protocol gateway enables bidirectional communication which makes it suitable for smart meters domain, because smart meters need a gateway that enables bidirectional communication, as it this very crucial functionality of smart meters [5]. The Azure IoT protocol gateway act as a bridge between an IoT hub and devices as shown in Figure 2.4 [22]. The Azure IoT hub is a cloud platform that supports securely connection and monitoring of millions of connected IoT devices [24]. IoT hub supports communication over a standard set of protocols which are MQTT (Message Queuing Telemetry Transport), Advanced Message Queuing Protocol and HTTPS (Hyper Text Transfer Protocol with Secure Sockets Layer) [5]. The Azure IoT protocol gateway was designed, because not all devices and sensors use these set of protocols. In general, the Azure IoT protocol gateway can be deployed on a cloud service as software gateway or in an on-premise environment, such as on field gateways which represent hardware gateways [5].

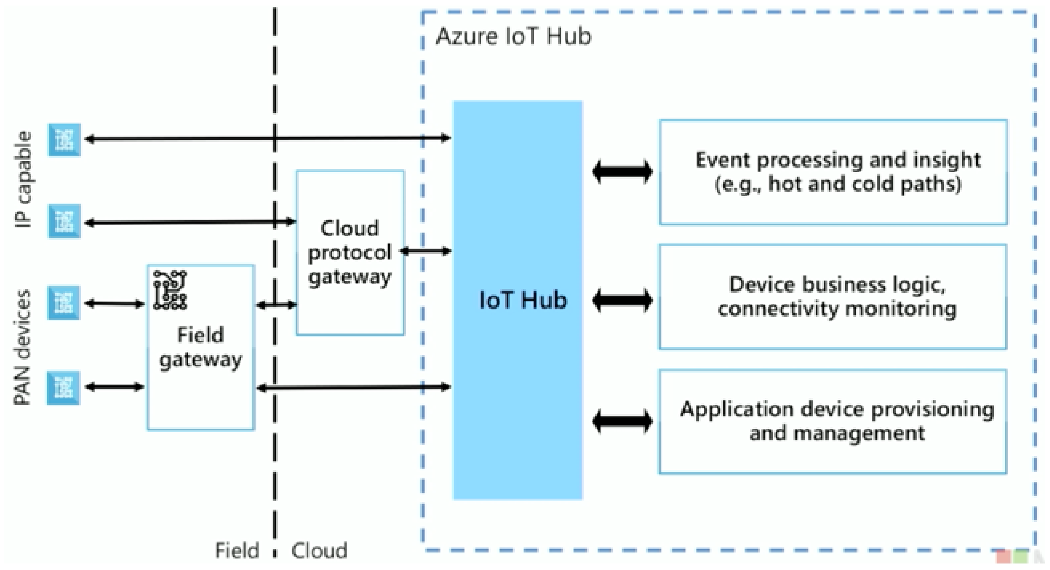

Azure IoT Hub
IP capable
PAN devices
Field gateway
Cloud protocol gateway
IoT Hub
Event processing and insight (e.g., hot and cold paths)
Device business logic, connectivity monitoring
Application device provisioning and management
Field
Cloud

Figure 2.4: Field Gateway and Cloud Gateway Communicating with IoT Hub [24]

The MQTT protocol is one of the most popular protocols for communication with machines in the IoT field. So in in order to customize the behavior of MQTT, the Azure IoT protocol gateway provides a MQTT protocol adapter. Both the Azure IoT protocol gateway and the MQTT adapter are open-sourced projects in order to provide flexibility. Therefore, the developer can use the open source software project to add a variety of not only protocols, but also protocol versions. In addition, we may create a custom design and implementation after modifying and adding modules to the open-source project for specific scenarios. The advantages of using IoT protocol gateway for protocol conversion are as follows:

- The protocol is highly scalable, as it can support millions of devices connected to it.
- The product enables custom authentication.
- The product enables message transformations.
- The product enables compression/decompression, for data exchanged between IoT hub and devices.
- Enables encryption/decryption of data exchanged between IoT hub and devices.
- The project is open source to add support various protocols.

There are numerous advantages of using Azure IoT protocol gateway as the gateway between cloud provider and devices. However, there are also disadvantages; one of the most important disadvantages of using the Azure IoT protocol gateway, is that the cloud provider has to be an Azure IoT hub, i.e, the gateway can only connect to Azure IoT hub in order to transmit the data transited by the devices. Sometimes developers will prefer using other cloud provider such as AWS or the Google Cloud platform. However, in general Azure IoT hub is similar to both of them.

Chapter 3

# Analysis of Protocols

The aim of this chapter is to introduce the protocols that are used in the thesis project. First, we will introduce an overview over the protocols DLMS/COSEM and ANSI 12.18 and their application layers. Then we will present a comparison between both DLMS/COSEM and ANSI 12.18, highlighting their advantages, common features, and differences.

## 3.1 The DLMS/COSEM Protocol

DLMS/COSEM provides an interface model, which supplies a view and a user interface for the functionality of the metering equipment, and a communication protocol which defines the method of data transportation and data access [9]. DLMS/COSEM consist of two parts, the DLMS part and the COSEM part. The DLMS (Device Language Message Specification) "is a generalized concept for abstract modeling of communication entities" [9]. It is the protocol used for communicating between meter equipments. On the other hand COSEM (Companion Specification for Energy Metering) characterizes a set of objects to interchange data with the metering equipment using DLMS protocol [9]. The physical meter is modeled as a logical device using COSEM; the logic devices has logical device name which is a unique identifier. Data in each logic device can then be accessed using interface objects, which are modeled using association objects. In a given context, the data about the available resources in a logic device are available using association objects [33]; this means that we can consider this logic device as a container that contains COSEM objects. These objects collect data such as power consumption, which are in the form of attributes and methods [6]. The availability of certain resource depends on the given access rights [33].

The information stored within a smart meter are in the attributes form, where the attribute values represent the specification. Each COSEM object in the meter equipment has a logic name as the first object as it acts as identification key for the object itself [33].

## 3.2 The DLMS/COSEM Application Layer

The information exchange between the meter and the data collector takes place between two application process (APs), which are the client and the server APs. The data exchange happens through protocol stack, which is shown in Figure 3.1. In the DLMS/COSEM architecture there are three layers, the application layer, the intermediate layer and the physical layer. Each layer provides a service to its upper level, while it receives a service from its lower level. In this section we describe the application layer of DLMS/COSEM.

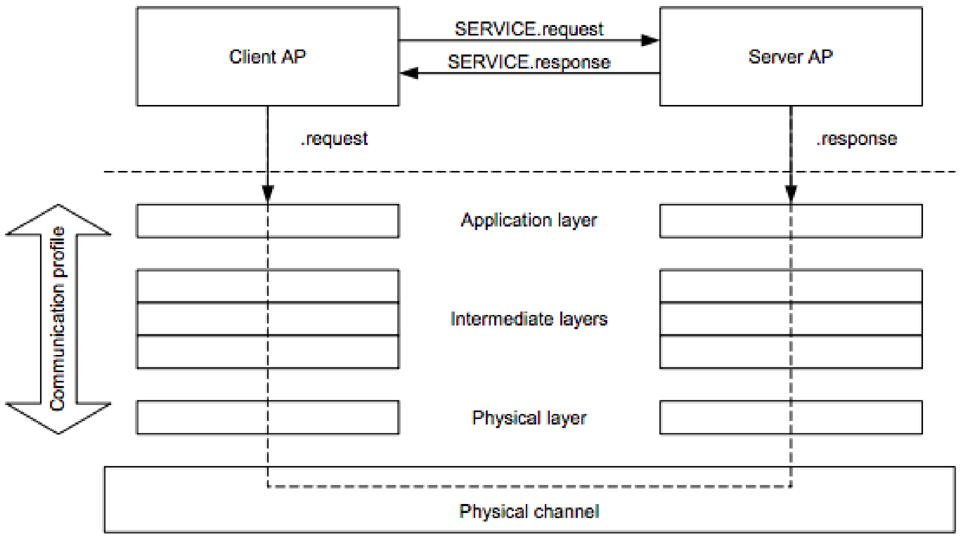


Figure 3.1: Client/server Protocol Stack (from [9])

The main part of the COSEM application layer (AL) is the COSEM application service object (ASO), which offers a service to the user, COSEM application process and service by the lower layers. It has three main components on the client and the server side. The first component is Application Control Service Element (ACSE). The task of ACSE is to create, maintain and release an application association. The next component is the extended DLMS Application Service Element (`xDLMS_ASE`), which is responsible for data transportation between the COSEM application processes. There exist two methods for reference in the DLMS/COSEM smart meters. One is the logic name and the other is the short name. Therefore, there exist two types of `xDLMS_ASE`, one that provides login name referencing, while the other offers short name referencing; it depends on the server of the application layer whether to have one type of `xDLMS_ASE` or the other or both together. The last component of ASO is the control function which specifies how ASO calls and invokes functionalities of the ACSE, `xDLMS_ASE` and the supporting layer services. The diagrams illustrated in Figures 3.2 and 3.3 describe the definitions of the client side and the server side control function. In these diagrams the ASSOCIATED state in both client and server models is the state where a connection is established and all functions of xDLMS are available for usage. The control function stays in this state until a release request is performed by the client.

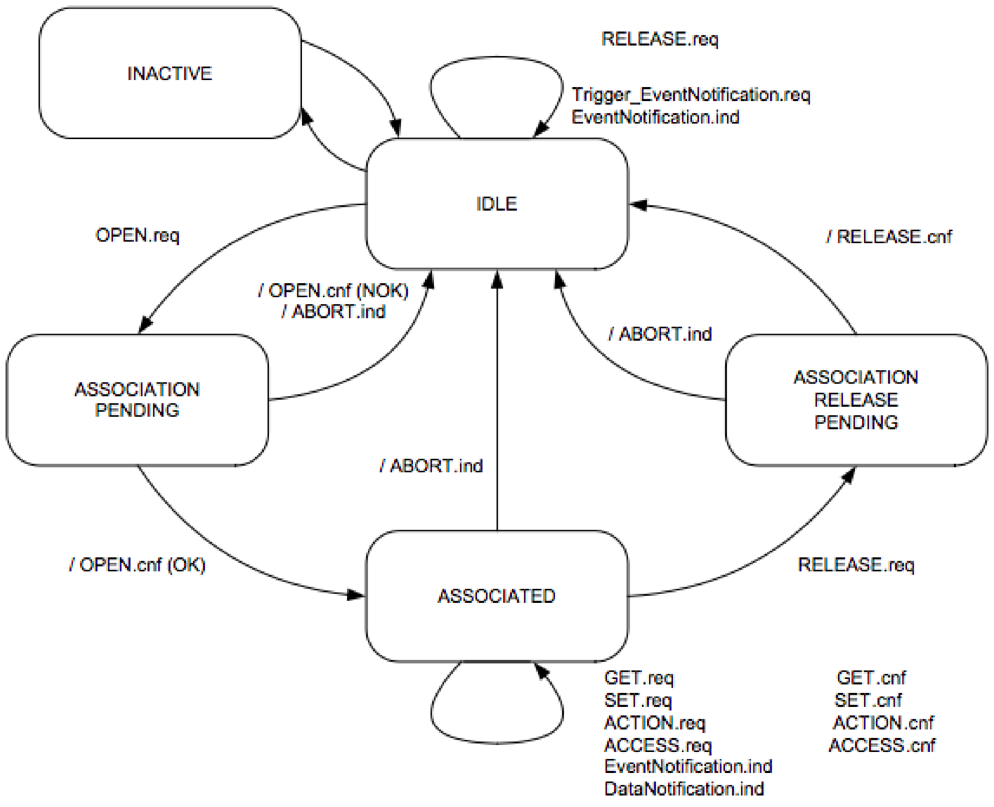


Figure 3.2: State Machine for the Client Side Control Function (from [9])

Diagrams 3.2 and 3.3 show five states for the control function in both the client and the server side. The first state is Inactive which is the state were no activity is happening in the control function. In the IDLE state no AA is created or released or being created for the client side; in this state the CF can handle the EventNotification service, while in the server side the CF can handle the EventNotification or InformationReport services. In the Association pending state, the AP has requested the creation of AA by `COSEM_OPEN` request. In the Associate state the AA has already been established; also the data service are available for data exchange in this state. The final state is Associate release pending, which is entered when the AP requests the `COSEM_RELEASE request`; the CF remains in this state till the server accepts the release request and then enters automatically the IDLE state again.

## 3.3 The ANSI C12.18 Protocol

ANSI C12.18 is a protocol that uses optical port communication to exchange meter data with utility provider so it is described using ANSI C12.19 [29], which is a standard for data exchange from and to the meter and utility provider using a table data structure. The table is divided and grouped in to certain sections. Each section maps to certain feature set and related functions, such as time of use. Data exchange occurs by reading and writing to a certain table or part of a table [8]. In order to exchange information and allow configuration and programming, the application layer provides services and data structures which are needed to support devices that use the ANSI C12.18 protocol. ANSI C12.18 has

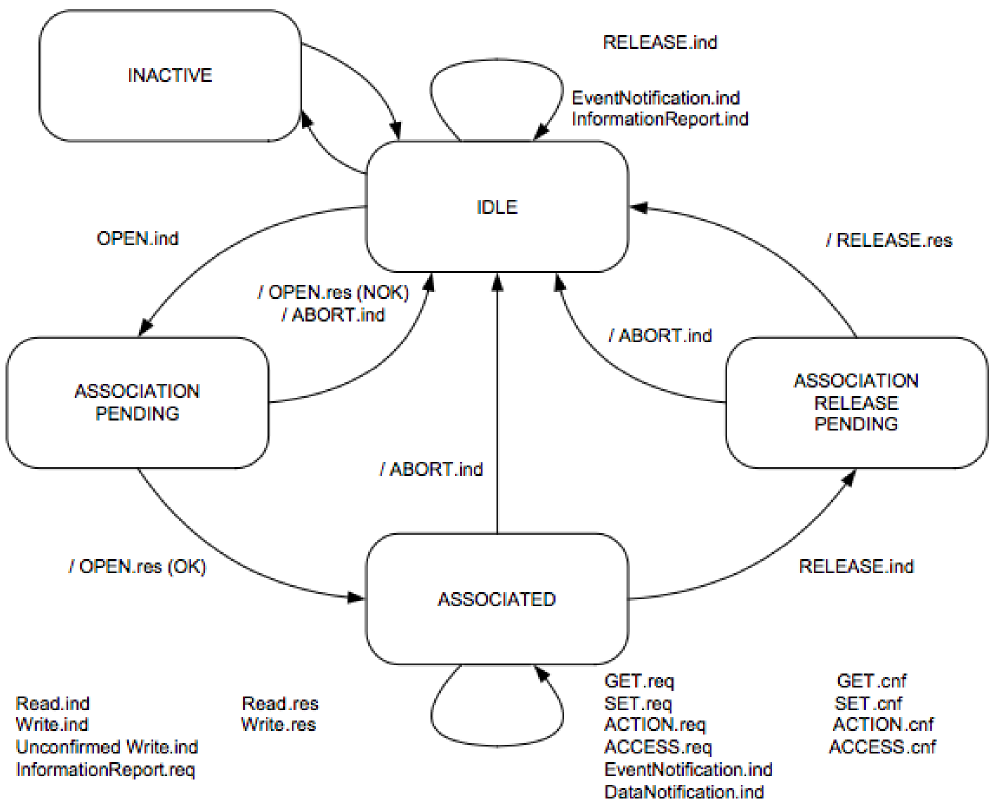


Figure 3.3: State Machine for the Server Side Control Function (from [9])

several applications not only in smart meters industry, but also for other applications, such as in smart street lightning.

## 3.4 The ANSI 12.18 Application Layer

The ANSI 12.18 protocol stack consists of seven layers shown below from the lower layer till the upper layer:

1. Physical layer.

2. Data link layer.

3. Network layer.

4. Transport layer.

5. Session layer.

6. Presentation layer.

7. Application layer.

The application layer is responsible for providing a group of services and a table data structure which is described by ANSI C12.19; it lets ANSI C12.18 devices be supported by the ANSI C12.18 protocol. The services provided by application layer are nine sets of request services and nine sets of response services. The list below shows the set of services for both request and response:

For the requests service:

- <**Ident**> Identification service request.
- <**read**> Read service request.
- <**write**> Write service request.
- <**logon**> Logon service request.
- <**security**> Security service request.
- <**logoff**> Logoff service request.
- <**negotiate**> Negotiate service request.
- <**wait**> Wait service request.
- <**terminate**> Terminate service request.

For the responses service:

- <**Ident-r**> Identification service response.
- <**read-r**> Read service response.
- <**write-r**> Write service response.
- <**logon-r**> Logon service response.
- <**security-r**> Security service response.
- <**logoff-r**> Logoff service response.

- <**negotiate-r**> Negotiate service response.
- <**wait-r**> Wait service response.
- <**terminate-r**> Terminate service response.

The initiation of a reading service or writing service is only done after opening a session, which is established using the Logon service. Using the read service, the table data is transferred to the requesting device; using the writing service, the table data is transferred to the target device, in our case the smart meter. After session is finished, it should be shut down using the Logoff request. The idle period in the application layer is maintained between communication devices using the Wait Service. The Terminate service request immediately shuts down the service and stops the communication channel.

## 3.5 Comparison between DLMS/CSOM and ANSI C12.18

In this section we are going to present the features of each protocol. In addition we are going to present the advantages and the disadvantages of using one protocol over the other.

### 3.5.1 DLMS/COSEM

DLMS/COSEM is one of the most popular protocols for smart meters communication in Europe. There are some features that makes DLMS/COSEM unique compared to other protocols. To begin with, DLMS/COSEM protocol was designed specifically for automatic meter reading [9], which means that the specification of the protocol as well as the design of the protocol is done in order to fulfill the smart metering required and recommended features. In addition, one of the most crucial advantages of using DLMS/COSEM is the ability for smart meters to exchange data and communicate to any data collection system, regardless of manufacturer of the smart meter or the type of the smart meter, such as whether it measures electricity, gas, or water. In addition, DLMS/COSEM smart meters offer a standard identification system, which means that objects with the same specifications in smart meter will have the same identification [32]. As the DLMS/COSEM protocol is used specifically for exchanging data with smart meters, it has an interface that is designed specifically for all kinds of energy types such as electricity, gas, and water [31]. Since the data is different from one energy type to another energy type, the interface for each energy type has a unique standard identifier. Moreover DLMS/COSEM allows manufacturers to innovate in order to always create evolution to the usage of the protocol by specifying specific instances, attributes and methods without changing the main structure of the identifiers [31].

Although DLMS/COSEM has many advantages, it also have some disadvantages over other protocols. Although the protocol enforce interoperability of the smart meters by creating standard identification system (the OBIS system), the former smart meters that used the DLMS/COSEM protocol did not have to use the standard identification system and manufacturers could define the identification system as they prefer. Therefore, former smart meters have interoperability issues when they communicate with a system that uses the standard identification system. For example the DLMS/COSEM smart meter that was used in this project did not use the standard identification system. Therefore, the generic API was developed in order to ensure interoperability. Another problem with DLMS/COSEM smart meters is high authentication, the reason is that no high level security mechanism is specified by DLMS/COSEM. As a result, manufacturers implements their own security mechanisms, which does not ensure high authentication with all smart meters that use DLMS/COSEM [13].

### 3.5.2 ANSI C12.18

ANSI C12.18 is a protocol that can work with several IoT applications, not just smart meters. The ANSI C12.18 protocol is very widely used in the United States of America. One of the biggest advantages of ANSI C12.18 smart meters is providing standard methods for communication. Specially for new designed smart meters the ANSI C12.18 standard specification is basic requirement. In addition smart meters using the ANSI C12.18 protocol tend to be adjustable as the usage of the meter change, such as high consumption customers such as factories. However, deploying such high scale smart meters is not economic for smaller usage, such as residential meters [29]. Another advantage is using the ASNI C12.19 table data structure, which enables very efficient data exchange between a smart meter and a data collection service in a home gateway using any type of physical transport with the meter, as ANSI C12.19 is fully extensible with additional data structures [29].

One of the disadvantages of the ANSI C12.18 protocol is that it can be used for multiple applications not just smart metering. This make some of its features not customizable to the Automated Meter Reading (AMR). In general there have been improvement of the ANSI C12.18 protocol, such as the ANSI C12.21 protocol and the ANSI C12.22 was implemented as the latest version of the ANSI protocols. ANSI C12.22 provides a better security mechanism as it offers both session and sessionless communication, unlike ANSI C12.18, which provides only session oriented communication. Therefore, ANSI C12.22 provides less signaling overhead [34].

Chapter 4

# Design of a Generic API

This chapter presents the generic API developed in this thesis. At the beginning, it describes the communication between a data collection service such as electricity company and smart meters through a gateway. This is followed by a description of the challenges that face developers in order to communicate with different smart meters that use different protocols and naming conventions. Then we describe the generic API in abstract way to overcome the challenges described and to provide protocol support no matter what type of protocol or naming convention used by the smart meter. At the end of the chapter, we describe the API using UML and we explain the supported methods in the API.

## 4.1 Communication between Data Collection Service and Smart Meters

The communication between a data collection service and smart meters is a bidirectional communication; the data collection service can read information from smart meters, such as power consumption in particular time, or it can write information on smart meters such as resetting the smart meters power consumption value register. The communication between a data collection service and smart meters usually proceeds via a gateway that serves as a bridge between the data collection service and the smart meter. The gateway can be either a hardware or software appliance. Gateways have many usages, for example data duplication to allow fault tolerance, or compression and cashing to make data translation faster or encryption to allow secure data protection. In our implementation the protocol conversion over the gateway allows communication and connectivity between both sides without depending of type of protocol used by different smart meters connected to the gateway. Figure 4.1 shows the connection architecture between data collection service and smart meters.

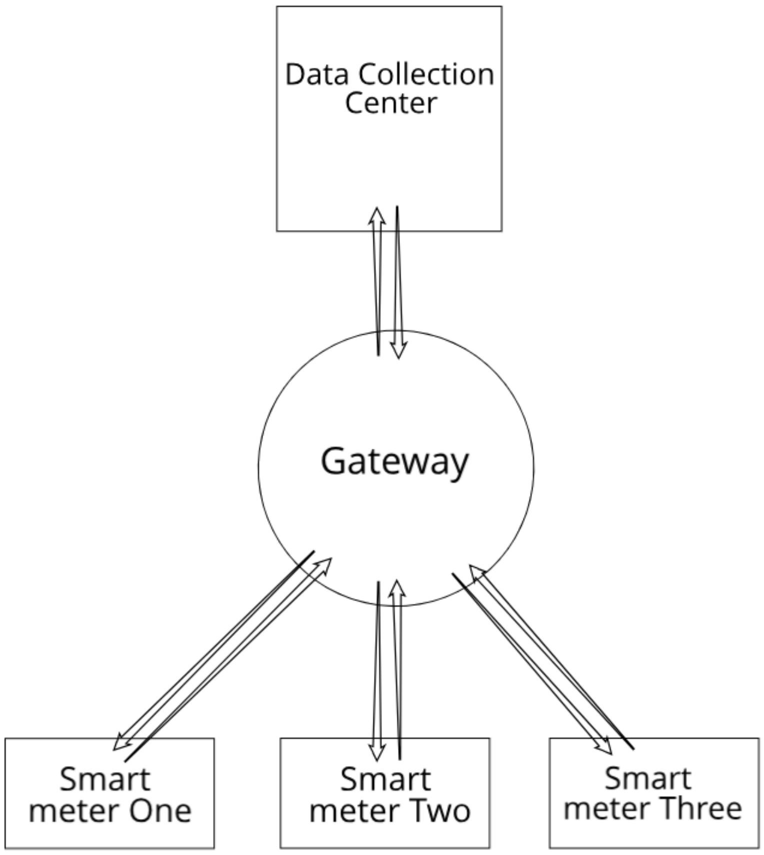


Figure 4.1: Communication between Data collection service and smart meters

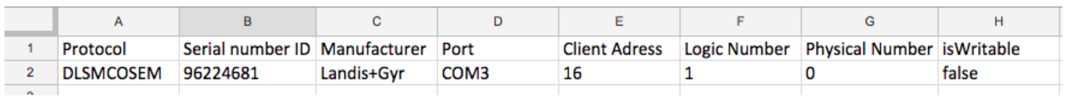
## 4.2 Interoperability Challenges in Smart Meters

Smart meters can vary in different parameters which adds challenge on the data collection service, because customers may have very diverse types of smart meters. The first and most important issue that can cause incomparability is the protocol used by smart meters; there is a wide variety of protocols and it depends on manufacturer which one to use. This variety of protocols add challenge in terms of parameters of initializing connection, data transfer or closing connection, as it is different in other protocols. Another difference is naming conventions, for example in the DLMS/COSEM protocol some smart meters use short naming conventions, while other use logic naming convention. Moreover, another challenge is different data types, in smart meters when reading a value of power from register it could be integer while in another one the returned value could be double. Finally, difference in units add challenge, for instance one meter may return a value of power in KW while other meter returns it in W, in other words there is no constancy in terms of units throughout different smart meters.

## 4.3 Gateway Design

In order to implement the generic API to translate protocols, we have to design and implement a gateway where our API can translate protocols over it. We used a computer as hardware gateway that act as middle man between smart meters and data collection service. Our gateway has a configuration file which is designed as Comma-separated values (CSV) file that includes information about the gateway and smart meters connected as follows:

Figure 4.3 shows CSV configuration file that has one DLMS/COSEM smart meter supported, which contains elements described in the list above in order.



| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Protocol | Serial number ID | Manufacturer | Port | Client Adress | Logic Number | Physical Number | isWritable |
| 2 | DLSMCOSEM | 96224681 | Landis+Gyr | COM3 | 16 | 1 | 0 | false |

Figure 4.2: CSV Configuration File with One DLMS/COSEM Smart Meter Supported

## 4.4 The API Specification

The API operations are expressed as abstract functions, subroutines or methods, which may be subsequently connected to language binding such as Java. The data types in the generic API are organized in Table 4.1. Table 4.1 shows an overview of the main building blocks of our API which defines different data types used by the system. The next subsection will describe each module of the API in detail.

### 4.4.1 Gateway Data Type

Our API enables the data collection service to communicate with gateways to get information about meters and gateways or write data into meters and gateways. Since a data collection service may communicate with multiple gateways, not only single gateway, our API has two methods to return gateways. The first method returns a default gateway, while the other one returns specific gateways by ID. Gateway is the main building block in our API, for example a data collection service can not get meters or get readings without specifying a gateway to return the data. In general, the gateways connected to the data collection service are in a collection which has the name gateway pool. The data collection service queries the gateway pool in order to return specific gateway. The operations for returning a gateway from the gateway pool, used by the gateway factory, is described in Table 4.2:

| **Data type name** | **Description** |
|---|---|
| Gateway | Represents the gateway in our system, can be a hardware gateway or a software gateway that acts as middle man between the meter and the data collection service. |
| Meter | Represents a meter in the system, a meter connects with one or more gateways. A meter contains registers that record information about the meter such as the manufacturer of the meter or some measured value of the meter. |
| Register | Represents the registers inside each meter that record measured values; registers can also it can contain abstract values about the smart meter. |
| Virtual protocol | Virtual protocol represent the OBIS standard, which standardizes not only the naming convention of the registers but it also standardizes the data types that are returned by querying a register. |
| DLMSProtocol | As DLMS/COSEM smart meters in the past used to work with different standards rather than OBIS the data type DLMSProtocol allows developers to use the API for that particular standard. This allows the user to query a register with the standard register name; the result will be data in the standard unit of the DLMSProtocol standard and in the standard data type of DLMSProtocol standard. |
| Register unit | A Register unit describes the unit of the data returned by a particlar register (for example W for power consumption unit Watt). |
| DataStore | A DataStore object represents the value of a measurement and type as a register type, both encapsulated into one object. |

Table 4.1: Overview of Main Building Blocks of the Generic API

| **Method name** | **Return type** | **Method Description** | **Throws Exception** |
|---|---|---|---|
| getGateway(string gatewayID) | Gateway | Returns specific gateway queried by the gatewayID. | Yes (GatewayNotFoundException) |
| getGateway() | Gateway | Returns the default gateway which is set by the data collection service which market as default gateway in the configuration file. | Yes (GatewayNotFoundException) |

Table 4.2: API Methods for Retrieving Gateway Objects

#### 4.4.1.1 getGateway(string gatewayID)

The method getGateway(string gatewayID) returns the gateway which is the initial building block of our API. The data collection service will communicate with gateways, as each gateway could have a distinct geographic location or the number of meters could be extremely huge such that the meters will be distributed among several gateways. Each gateway is given a unique sequence of alphabetic and numbers identifier, which is stored in the configuration file. The method getGateway(string getewayID) fetches all gateways and returns a gateway which has the configuration file gateway identifier as the parameter getewayID. The collection of gateways that are connected to the data collection service is called gateway pool. A gateway pool represents the set of gateways which can be queried by the data collection service by getGateway(string gatewayID) to return a specific gateway.

#### 4.4.1.2 getGateway()

The method getGateway(), is different from getGateway(string gatewayID) as it takes no arguments; it returns the default gateway. The data collection service or the utility provider is responsible for defining one and only one default gateway. For example, the data collection service could define a default gateway for a gateway that is connected to the test meters, or it could define a default gateway for a gateway that has the largest number of meters connected to it. The configuration file contains one boolean variable isDefault which defines if the gateway associated to the configuration file is default gateway or not. If the data collection service defined several default gateways as default gateways, only the gateway marked as default that has the least lexicography gatewayID is returned, (the algorithm searches every gateway in the order of the gatewayID and once it finds a gateway in the configuration file that has isDefault of true, it returns that gateway).

### 4.4.2 Gateway Methods

The gateway allows on data collection service or utility provider to query different meters connected to it. This helps the data collection service to track the different meters which are used by the customers. The gateway supports four different methods for retrieving meters, or information about the gateway. The supported methods are described in Table 4.3.

#### 4.4.2.1 getMeter(string meterID)

The method getMeter(string meterID) is invoked on a gateway object; it returns an object of type meter. Each meter is assigned by unique meterID identification string. The gateway is aware of each meter connected to it, because of the configuration file. The configuration file contains data about each meter and the details about each meter such as the meter identifier. The method iterates over all meters

| Method name | Return type | Method Description | Throws Exception |
|---|---|---|---|
| getMeter(string meterID) | Meter | Returns a specific meter queried by the meterID. | Yes (meterNot-FoundException) |
| getAllMeters() | List <Meter> | Returns the list of meters connected to the gateway. | No |
| getAllMeters-ByProto-col(Protocol protocol) | List<Meter> | Returns a list of meters that support a specific protocol "protocol" | No |
| getAllSupported-Protocols() | List <Protocol> | Returns the list of protocols that are supported by the gateway. | No |
| getGatewayID() | string | Returns the string identification value of the gateway. | No |

Table 4.3: Gateway Methods

that are connected to the gateway and returns the meter with the same meter identifier as the method parameter.

#### 4.4.2.2 getAllMeters()

The method getAllMeters() returns all meters connected to the gateway as a list of meter objects after being invoked on a gateway object. The list of meter objects is retrieved by iterating over the configuration file and returning all meters in the configuration file in the list form. Whenever a new meter is connected to the gateway it must be added to the configuration file so that getAllmeters() can add this new meter to the list of meters.

#### 4.4.2.3 getAllMetersByProtocol(Protocol protocol)

The method getAllMetersByProtocol returns list which has meters objects that use a certain protocol. The method cause its configuration file to find the meters that use the same protocol as the one in the method parameter. Meters with the same protocol will have similar properties such as naming conventions, register types and units.

#### 4.4.2.4 getAllSupportedProtocols()

The method getAllSupportedProtocols() return a list of protocols, these protocols are supported by the generic API. These protocols are returned from the configuration file where it is described which protocols the system supports. Therefore a user can add many meters in the system as long as the protocols of those meters are supported with the protocols returned from the list getAllSupportedProtocols(). The protocols supported by getAllSupportedProtocols() have specific functions and methods for the supported protocols handling. When a new protocol is supported, it should be added to the supported protocols in the configuration file.

#### 4.4.2.5 getGatewayID()

The method getGatewayID() returns the gateway identifier which is a string identification value. Getting the gatewayID which is recorded in the configuration is useful when the data collection service has a large number of gateways and developers need to know the gatewayID of a particular gateway. This is especially important for default gateway, where the user does not have to specify the default gateway with the method getGateway(string gatewayID) and can only specify it with method getGateway().

### 4.4.3 Meter Methods

The meter object represents physical meter connected to the data collection service through gateway. As mentioned before one or more smart meters are connected to a gateway while a smart meter has many registers. Registers are distinct from one smart meter to the other. The meter object supports several methods which are used no matter what protocol is used. For example for a read function it can be called over a DLMS/COSEM protocol smart meter or an ANSI C12.18 smart meter. Therefore, a level of abstraction is introduced that enables developers to use the system no matter what type of protocol used by the smart meter, type of meter or the manufacturer of the smart meter. However, the generic API system supports only the protocols that are added to the configuration file. The methods supported by the meter object in our system are shown in Table 4.4.

#### 4.4.3.1 getAllRegisters()

The method getAllRegisters() returns all the registers within the smart meter object invoked upon the method. The returned object is a list of register objects which is described in Subsection 4.4.4. The generic API iterates through the registers used in the system to generate the list. The process of iterating through the registers is done depending on the protocol used by each meter. The registers naming convention used by a smart meter that uses a certain protocol is different than another smart meter that uses a different protocol.

#### 4.4.3.2 getManufacturer()

The method getManufacturer() returns the manufacturer of the smart meter. This helps as some smart meters have different specifications due to manufacturer preference. For example the DLMS/COSEM

| Method name | Return type | Method Description | Throws Exception |
|---|---|---|---|
| getAllRegisters() | List<Register> | Returns all registers that the smart meter has. | No |
| getManufacturer() | string | Returns the manufacturer name of the smart meter. | No |
| getProtocol() | protocol | Returns the protocol that is used by the smart meter. | No |
| getRegister(string registerID) | register | Returns the register with the identifier registerID. | Yes (RegisterNot-FoundException) |
| getMeterID() | string | Returns the string identification value of the meter. | No |
| hasRegister(string registerID) | boolean | Returns whether the smart meter contains a register with register identifier registerID | No |
| openConnection() | void | Starts connection between the gateway and the smart meter. | Yes (IOException) |
| closeConnection() | void | Closes the connection which was opened to prevent further communication between smart meter and gateway. | Yes (IOException) |
| isWritable() | boolean | Returns whether the smart meter allows writing to it. | No |
| disableManual-Disconnect() | void | Disables the owner of smart meter to turn it off. | Yes (IOException and NotWritableMeterException) |
| enableManual-Disconnect() | void | Enables the owner of smart meter to turn it off. | Yes (IOException and NotWritableMeterException) |
| breakerClose() | void | Reaches the closed state which enables the smart meter to function and opens electricity. | Yes (IOException and NotWritableMeterException) |
| breakerOpen() | void | Reaches the locked open state which turns off the electric connection, causing the connected object to be turned off. | Yes (IOException and NotWritableMeterException) |
| breakerUnlock() | void | Reaches the open state which returns the string identification value of the meter. | Yes (IOException and NotWritableMeterException) |

Table 4.4: Meter Methods

naming convention could use a logic name, a short name or both. The use of a naming convention over the manufacturer name and details regarding which naming convention is used is recorded in the configuration file. The method getManufacturer() returns the manufacturer name from the configuration file.

#### 4.4.3.3 getProtocol()

The method getProtocol() is invoked on a meter object to return the protocol used by the smart meter. The method returns the protocol object that represents the type of protocol used by the smart meter. This is particularly important because some of the methods work on a certain protocol would not work on another protocol, so developers may check if the specific protocol would be compatible with a specific method without throwing an exception.

#### 4.4.3.4 getRegister(string registerID)

One of the key methods that could be invoked on the meter object is getRegister(string registerID) as it returns a specific register which has the register identifier registerID parameter. The method returns the register object which is defined in Subsection 4.4.4 This method is particularly important as developer can use the returned register object for reading or writing on it. The method iterates over the list that represent all registers within the smart meter; once the register with same identification as the registerID parameter is found, the current register in the list is returned.

#### 4.4.3.5 getMeterID()

The method getMeterID returns the meter identifier of the meter object that is invoked on the method. The meter identifier is a string representing the serial number of the meter which is stored in the configuration file. The meter identifier and the meter manufacturer are used to define meters as every meter has its own unique serial number and manufacturer combined. In other words the serial number could be the same across different manufacturers. However, the same manufacturer does not manufacture two smart meters with the same serial number.

#### 4.4.3.6 hasRegister(string registerID)

When the method hasRegister(string registerID) is invoked on an object meter with parameter registerID, it returns as a boolean value whether the smart meter contains such a register. This method is particularly important to check if the naming convention will return register that is defined in our system as it is within the invoked meter. When calling method hasRegister(string registerID) the returned true false value allows checking if a register is within a meter before reading from that register; this ensures safety rather than receiving an exception for not found register inside the smart meter.

#### 4.4.3.7 openConnection()

One of the fundamental methods that meter a object invokes is openConnection(). The method openConnection() opens the connection to allow the gateway to query registers; it allows reads and writes to the smart meter. openConnection() initiates the connection which depends on the type of the protocol used by the meter. For example DLMS/COSEM uses different parameters to initiate a connection compared to ANSI C12.18, in other words DLMS/COSEM connection is completely different from the way ANSI C12.18 connects the gateway to the smart meter. Therefore the way of initiating the connection is different. However our generic API is smart enough to change internally the way of connection depending on the protocol. The different parameters for initiating a connection are stored in the configuration file to allow the generic API to initiate a connection without the developer caring about the underlying protocol.

#### 4.4.3.8 closeConnection()

The method closeConnection() is very similar method to the method openConnection(). After initiating a connection by the method openConnection(), the connection should be closed after all the data exchange has happened. Similar to openConnection(), closeConnection() closes a connection depending on the underlying protocol used. For example, the DLMS/COSEM protocol has different parameters for closing of connection compared to ANSI C12.18. Closing a connection is crucial, as without closing connection, errors and exceptions could occur as unintentional reads and writes could happen, so it is safer to close connection. The other reason is that if the gateway kept the connection open between every meter connected to it, this will lead the gateway to become slower due to overloading the system.

#### 4.4.3.9 isWritable()

The method isWritable() returns a boolean value of whether the invoked meter is writable or not. The information related if the meter isWritable() or not is stored as a boolean value in the configuration file. Each meter in the the configuration file has a value of whether it is writable or not. A meter may become not writable because of authentication privileges that are not obtained. The value of true or false of the method isWritable() help the developer to use the writing methods on the smart meter after checking if the meter is writable or not. If the meter is not writable and the developer invoked a writing method on the smart meter it will throw exception. Therefore, the developer should check if meter writable first in order to ensure safety of the methods used.

#### 4.4.3.10 disableManualDisconnect()

The method disableManualDisconnect() is a method that returns nothing. This method is a write method which can only execute when the meter is writable. Invoking disableManualDisconnect() on the meter object, lets the owner or the user of the smart meter become unable to disconnect the smart meter. Disconnecting smart meter means that the smart meter cuts the electricity on the electric suppliance using the smart meter. Therefore, the meter does not record the values as the electric suppliance is off, which leads the smart meter to be disconnected. The method disableManualDisconnect() forces the user or the owner of the smart meter not to disconnect the meter using the physical disconnect button on the smart meter. In other words, disableManualDisconnect() turns off the off button of the meter. In the ANSI C12.18 smart meter that we used throughout the project the off button had the symbol [ 0 ] which is disabled by the method call.

#### 4.4.3.11 enableManualDisconnect()

The method enableManualDisconnect() is a writing method which is the opposite of the method disableManualDisconnect(). It enables the user or the owner of the smart meter to manually disconnect the smart meter and make the smart meter cut the electricity usage by pressing on the physical button on the smart meter for disconnecting. The user or the owner of the smart meter might find it useful to disconnect the smart meter when he is leaving house for a period of time such as in vacation time. This ensures the user that the electricity will not be used through this time; therefore it ensures safety in terms of electricity usage and waste of energy. In other words, enableManualDisconnect() turns on the off button of the meter.

#### 4.4.3.12 breakerClose()

The method breakerClose() is invoked on the meter object by the data collection service in order to turn on the smart meter and allow the meter to provide electricity and measure the electricity consumption. Diagram 4.4.3.12 shows the different states of meter. The states of a meter is controlled by the data collection service and depending on the situation the data collection service can turn off or turn on the smart meter. The state marked closed in diagram 4.4.3.12 is the only state which allows the electricity to be connected to the meter and allows the measure of the consumption. In this state on the ANSI C12.18 meter that we used for experimenting, the light of the meter is turned off which indicates that the meter is working. This method has a particular use case when the user returns from a period where he was not using the electricity, such as when the user is on vacation away from home.

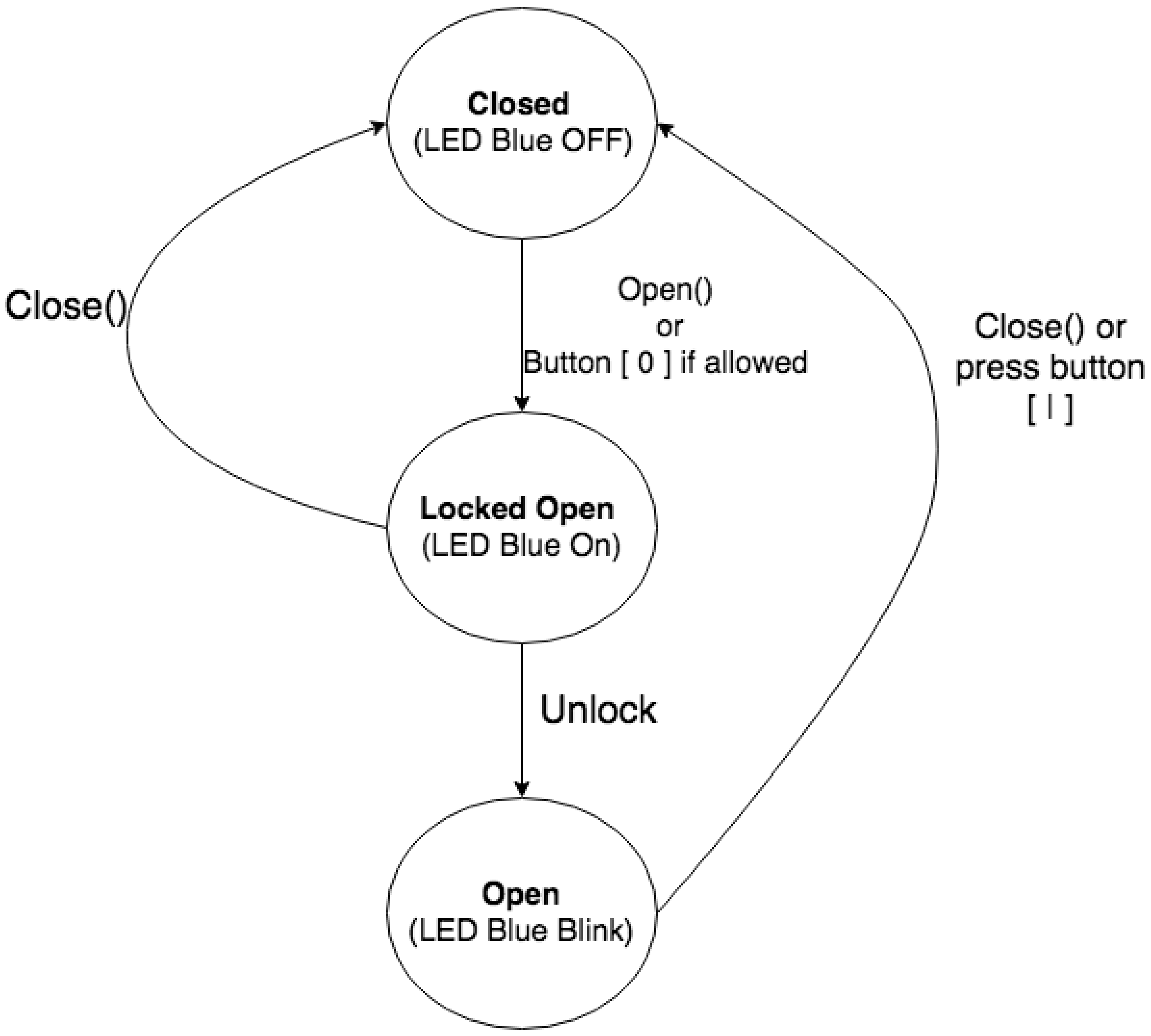


Figure 4.3: State Diagram Showing Different States of Meter

#### 4.4.3.13 breakerOpen()

The second method that is used to navigate in the state diagram is breakerOpen(). When calling the method breakerOpen() at the state of the being closed, in other words the smart meter is working, turns the smart meter off. In addition, it goes to state locked open. Therefore, the electricity becomes disconnected from the electric suppliance. In the state locked open the LED blue on the ANSI C12.18 meter becomes illuminated with the blue color as shown in digram 4.4.3.12. The state locked open could be reached also by the user if he pressed on the off button, which has the symbol [ 0 ] in the ANSI C12.18 smart meter. This way the user turns his own smart meter by moving the smart meter state from closed to locked open. However the only way for the user to turn the smart meter off by pressing [ 0 ] button is that the data collection service has allowed the user to do so. This can only be done in our system by the data collection service where it calls the method enableManualDisconnect() discussed earlier. Calling the method breakerClose() from the state locked open as shown in Figure 4.4.3.12 will move the smart meter state from being locked open where it is not functioning to the state closed where it is functioning and recording the consumption.

#### 4.4.3.14 breakerUnlock()

The final method in the state Diagram 4.4.3.12 is breakerUnlock(). The method breakerUnlock() moves the state of the smart meter for the state locked open to the state open. For the smart meter the method being in state open means the smart meter is not functioning and is not recording measurements. As shown in digram 4.4.3.12 the only difference between the open state and the locked open state is that the user in the open state can make the meter start recording the measurement and provide electricity by pressing on the on button which in our ANSI C12.18 meter has the on button represented by the button [ 1 ]. The state open has the smart meter LED illuminate with blue color but blinking. As mentioned no only can user press on the on button [ 1 ] to move from state open to close but also the data collection service can use the method breakerClose() to move to closed where electricity is provided and electricity measurement occurs.

### 4.4.4 Register Methods

The register object represents the registers that exist within a smart meter. A register object is identified by register identifier that is defined by the manufacturer, which is set based on the naming convention and the protocol type. Registers can have different specifications, types and units. Different specifications means that one register could be used to give data related to the smart meter used such as voltage transformer ratio, it could have measured data by the smart meter such as positive active energy (A+) total or it can have registers that allow writing. Register can have different data types, for example one register may have a double value stored in a register while another register may have long, integer, string, date or boolean value. Finally another distinct parameter between registers are register units. Registers may have different units based on the information that the register is measuring. Kw, W, kWh, kvarh, Wh, varh and seconds are possible units for registers. Some registers do not have units for example boolean variables, so we assign it unit as none. RegisterType and RegisterUnit are two enums in our generic API that allow user to use the legal values for register type and register unit. Table 4.5 shows the methods supported by a register object in our generic API.

| **Method name** | **Return type** | **Method Description** | **Throws Exception** |
|---|---|---|---|
| getRegisterID() | string | Returns the string identifier for the register. | No |
| getRegisterType() | registerType | Returns an object of type reig-sterType which represents the type of register used. | No |
| getRegisterUnit() | registerUnit | Returns object of type regis-terUnit which represent the unit of the data in the register used. | No |

| | | | |
|---|---|---|---|
| open() | void | Opens register for reading or writing on it. | Yes (ConnectionException, IOException and InterruptedException) |
| close | void | Closes open register for after reading or writing on the register. | Yes (ConnectionException and IOException) |
| read() | list<DataStore> | Returns a list which represent the value read from the register along with the register meta data. | Yes (ConnectionException, IOException, ParserConfigurationExcpetion and InterruptedException) |
| getMeter() | Meter | Returns the meter that the register belong to. | No |
| timeReads() | double | Returns a number that calculates the time to read a register in the smart meter. | Yes (IOException) |
| readDouble() | double | Returns the double value of the register, if it has a double value. | Yes (IOException, InvalidDataTypeException and InterruptedException) |
| readInteger() | integer | Returns the integer value of the register, if it has an integer value. | Yes (IOException, InvalidDataTypeException and InterruptedException) |
| readBoolean() | boolean | Returns the boolean value of the register, if it has a boolean value. | Yes (IOException, InvalidDataTypeException and InterruptedException) |
| readLongInteger() | long | Returns the integer value of the register, if it has a long value. | Yes (IOException, InvalidDataTypeException and InterruptedException) |
| readString() | string | Returns the value in the register as a string. | Yes (IOException, InvalidDataTypeException and InterruptedException) |
| readDate() | date | Returns the date value of the register, if it has date a value. | Yes (IOException, InvalidDataTypeException and InterruptedException) |

Table 4.5: Register Methods

#### 4.4.4.1 getRegisterID()

The method getRegisterID() returns the register identifier string. Each register has a specific unique identifier. The method is particularly useful when working with a list of all registers; it needs to identify the register by the identifier, because each register has its own properties.

#### 4.4.4.2 getRegisterType()

The method getRegisterType() returns an object with the type registerType which represent the type of the measured value of the register. The object registerType is an enum object which has all the supported types by our generic API: string, integer, long, double, float, date, boolean and none. The none type belong to registers that do not have values as read value such as write registers. Each register has only one type of measured data. The method getRegisterType() helps developers know the type of data expected from the register; this helps the developer select one of the methods related to the data type such as readDouble or readBoolean explained in the upcoming subsections.

#### 4.4.4.3 getRegisterUnit()

The method getReisterUnit() returns an object with the type registerUnit which represents the unit of the measured value of the register. The object registerUnit is an enum object which has all the supported units by our generic API. Namely Kw, W, kWh, kvarh, Wh, varh, seconds and none. Some registers has none unit as boolean registerType does not have a numeric value for a unit. Therefore, none unit is given for the registerUnit. Each register has only one unit of measured data.

#### 4.4.4.4 open()

The method open() is invoked on a register object to open this register to allow reads and writes over the register. The method open() is extremely important for writing and reading without invoking the method before either operations will throw exception as register is not ready for both operations.

#### 4.4.4.5 close()

The method close() is invoked on a register object to close this register after the read or writes in the system. The close() method can not be invoked on the register object unless it was open and not closed. Closing a register offers safety for the developer as accidental writes could occur; also keeping the registers open cause a performance depredation of the gateway.

#### 4.4.4.6 read()

One of the most important methods that can be invoked over the register object is the read() method. The method returns a list of DataStore values.

DataStore is an object that is defined by the generic API, specifically to handle the interoperability of the data. The object DataStore is used to describe the details of the measured or abstract value in the register. DataStore consist of two attributes, an object and a regsiterType. The object represents the value of a measured or an abstract value. The regsiterType is the type of the data inside the register explained in Subsection 4.4.4.2. For example, for power consumption the DataStore could be represented as a value object 220.0 and registerType double. The object attribute of the DataStore can have a structure value. Structure value represent a new DataStore and the type of this structure value is STRUCTURE; in other words DataStore can have another DataStore inside it as object which allows nesting of DataStore.

The read method returns a list of DataStore values, because sometimes there are other data within a register that are not related to the measured value that need to be returned with the measured value; these are the abstract values. These values are returned are the scalar unit which is multiplied by the measured value for calculation of tariff, the time that the register was read, or the last average value of the current register that we are reading the value from. Therefore, the returned list gives insight to the developer about the type of register with the measured value along side with some information regarding the register (such as the last average value or the current average value or the last time the register was read).

#### 4.4.4.7 getMeter()

The method getMeter() returns a meter object that is explained in Subsection 4.4.3. The relationship between meters and registers is explained in Figure 4.4.4.7, where register is in one and only one meter while meter has many registers. The method getMeter() is particularly important when performing parallel reading on same register number but of different meters, so that each measurement is mapped to a certain meter which is modeled in our generic API as the meter object.

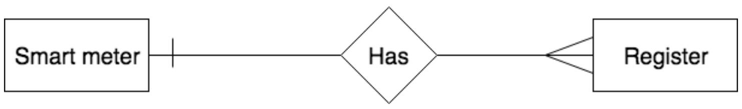


Figure 4.4: Relationship between Smart Meter and Registers

#### 4.4.4.8 readDouble()

The method readDouble() is one of the set of methods that reads the value of register by knowing its type in advance. For example if the developer is sure that a certain register has double type and call the method readDouble() over it, it will return a double value of the current measurement that is written in the register. In case that readDouble() is invoked on a register that does not have double value, an exception will thrown. With the help of method getRegisterType(), this exception could be avoided as a developer can check if the value stored in the register is double or not. If the stored value is not double then the developer can consciously choose the method that fits the data type.

#### 4.4.4.9 readInteger()

The method readInteger() is another method of the set of methods that reads the value of register using its type. The method readInteger() is similar to method readDouble(), however the difference comes to the accuracy of the number. For example, if one smart meter uses a different unit convention such as kW, while another smart meter uses W, this causes the type of the data to change, because kW needs more accuracy by having a double number instead of an integer. However, the smart meter using W will have the a register with type integer as it offers enough accuracy. In general developer can run readInteger() on double variable but it will cause a loss of accuracy which is unacceptable as it affects the billing of the customer.

#### 4.4.4.10 readBoolean()

Another method for reading the value of register using its type is readBoolean(). readBoolean() is used to read a true, false data from register. Such registers offers yes or no values to query. An example of such register "is Fraud" flag register that is flag with either true or false value. Method readBoolean() can not work with any other data type but boolean, as it casts the yes and no values to true and false respectively.

#### 4.4.4.11 readLongInteger()

The method readLongInteger() reads the numeric value of the register and casts it to long data type. The difference between readLongInteger() and readInteger() is that readLongInteger() reads the value up to 64 bit number while readInteger() reads the value up to 32 bit number. Therefore, if the number is bigger than 32 bit readLongInteger() should be used over readInteger() as readInteger() will not be able to store this big number. However, if the number is smaller than 32 bit and readLongIntger() is used, the number will be cast to long which will not affect the number, but it will occupy more memory as 64 bits are reserved for a smaller number.

#### 4.4.4.12 readString()

Another method for reading the value of register using its type is readString(). The method readString() reads any type of value stored in the register and casts it to string. ReadString() is the most general method as no matter type of value in register is, it will not throw exception as any type of data can be casted to string. However if the value is integer in the register and readString() is used, then this integer value will be stored as an integer, which will not be easy for computation as a number can not be processed as a string data type.

#### 4.4.4.13 readDate()

The final method for reading the value of register based on its type is readData(). The method readDate() reads the date value of the register in the form DD.MM.YYYY. In other words the date value represents the day, month and year stored in a register. One register that use the date data type is the "Date" register which stores the current date of querying the meter.

### 4.4.5 Protocol

The protocol object is used to model the protocol used by the meter. Each smart meter has a specific protocol that is used to initiate and close connections. In addition, it is used to read and write data with the gateway. In our generic API two protocols are supported. However, the the generic API is scalable and allow different other protocols to be added to the supported list. In general, the supported protocols are recorded in the gateway configuration file and can be returned by the method getAllSupportedProtocols() that can be invoked on gateway object. The two protocols that are supported by our generic API are DLSM/COSEM and ANSI C12.18. In order to support both protocols two different objects are extended from protocol object, specifically for each type of protocols. The two objects are VirtualProtocol and DLMSProtocol. Both protocols have similar objects that it can call. For example, VirtualProtocol has constant `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` which is the identifier of the register related to positive active energy; however the DLMSProtocol has the same `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` constant, which has identifier of the register related to positive active energy, but different value than the VirtualProtocol, as they use different naming conventions.

#### 4.4.5.1 VirtualProtocol

The VirtualProtocol object has the values for the OBIS code convention. Therefore, Virtual-Protocol is the object that covers the general case, where any protocol that uses OBIS convention is supported. OBIS offer a distinct identifier for all data within the smart meter, register types and register units; therefore our VirtualProtocol contains constants of each register Identifier as name such as `POSITIVE_ACTIVE_ENERGY_TOTAL_ID`, but it also contain the OBIS units and types which are also stored in constants such as `POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT` and `POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE`. The constants `POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT` and `POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE` are mapped to the registerUnit data type and registerType data types respectively.

#### 4.4.5.2 DLMSProtocol

The DLMSProtocol object represents the naming convention and the data types used by the registers in the smart meter that we used in our experiment that use the DLMS/COSEM protocol; however it does not use the OBIS code. Like VirtualProtocol, DLMSProtocol has constants that represent the identifier of the register, the type of register and the register unit. The information regarding this particular meter which was used in testing, such as its own naming convention was applicable and available by looping over all registers in the smart meter using the JDLMS library which is explained in Chapter 5. After looping over the smart meter, we are able to deduce the register name, type and unit and can store them in constants (similar to VirtualProtocol with the same constants naming convention to ensure unity of calling methods).

### 4.4.6 DataStore

DataStore object mentioned in the previous subsections is data structure for storing both data value measurement and its type. In other words both value and type are encapsulated into one object, which help the developer to use both variables as one object, this way it abstracts the dependence on type that is forced by methods, such as readDouble() and readInteger(). An example of dataStore object for power consumption when running it on the ANSI C12.18 smart meter is [double 230.2] which indicate that the data object has a double value of 230.2. The data structure data object supports all different register types that are maintained by the registerType object that was explained earlier.

## 4.5 Generic API Scenarios

In this section we are going to discuss two different situations that describe legal sequences interaction of the various operations supported by our generic API. Each situation will describe a situation where the developer will use the API; also it will describe the sequence of using the methods by a sequence diagram and by an exact description of using the API methods discussed in the previous sections.

### 4.5.1 Reading Registers

Situation one, described by the sequence diagram at Figure 4.5, is a situation where the data collection service, modeled as the actor in the sequence diagram, tries to read the positive active energy of a specific smart meter. The sequence diagram shows the interaction of the data collection service with each model object, in order to reach the goal intended by the data collection service. The sequence is arranged in time where some methods calls by the data collection service have returned values while other methods do not return a value. In the described situation, the data collection service is querying a ANSI C12.18 smart meter where it uses the OBIS code convention.

The situation starts by the data collection service querying the gateway pool, where all different gateways exist, by the method getGateway(), which returns the default gateway that is configured at the configuration file as the default gateway. If there are defined several default gateways in the gateway pool, the one with least gateway identifier will be returned. Calling the method getGateway() returns a gateway object to the data collection service. Next the data collection service can invoke the function getMeter("9963218"), which returns a meter object to the data collection service. The method getMeter("9963218") in the sequence diagram in Figure 4.5 is communicating with the gateway object created by owner generic API. The string "9963218" represents the identifier of the meter that the data collection service wants to read data from. After the meter object has been returned from querying the gateway, the data collection service applies methods on the returned meter object. The first is the openConnection() method which is very vital to start data exchange with the smart meter. The method openConnection() uses some information regarding the protocol used by the smart meter which is recorded in the configuration file, also the parameters that will be used in the connection are recorded in the configuration file. In the situation described, the smart meter uses an ANSI C12.18 smart meter which has a similar protocol and naming convention as the physical smart meter we used for testing. Parameters such as protocol, client address, port number as well as the writing directory of the smart meter readings are all recorded in the configuration file and are used to initiate the connection with the smart meter. After initiating the connection with the smart meter, the data collection service can query the smart meter with those registers it needs. However, if the connection could not be initiated, an exception will be thrown; this happens if there is problem with the physical connection with the smart meter or if there is a huge delay; in general connection problems depend on the method of connection of the smart meter with the gateway, e.g serial connection or connection similar to TCP/UDP connection. The method openConnection() is a void method; therefore it did not return any object to the data collection service.

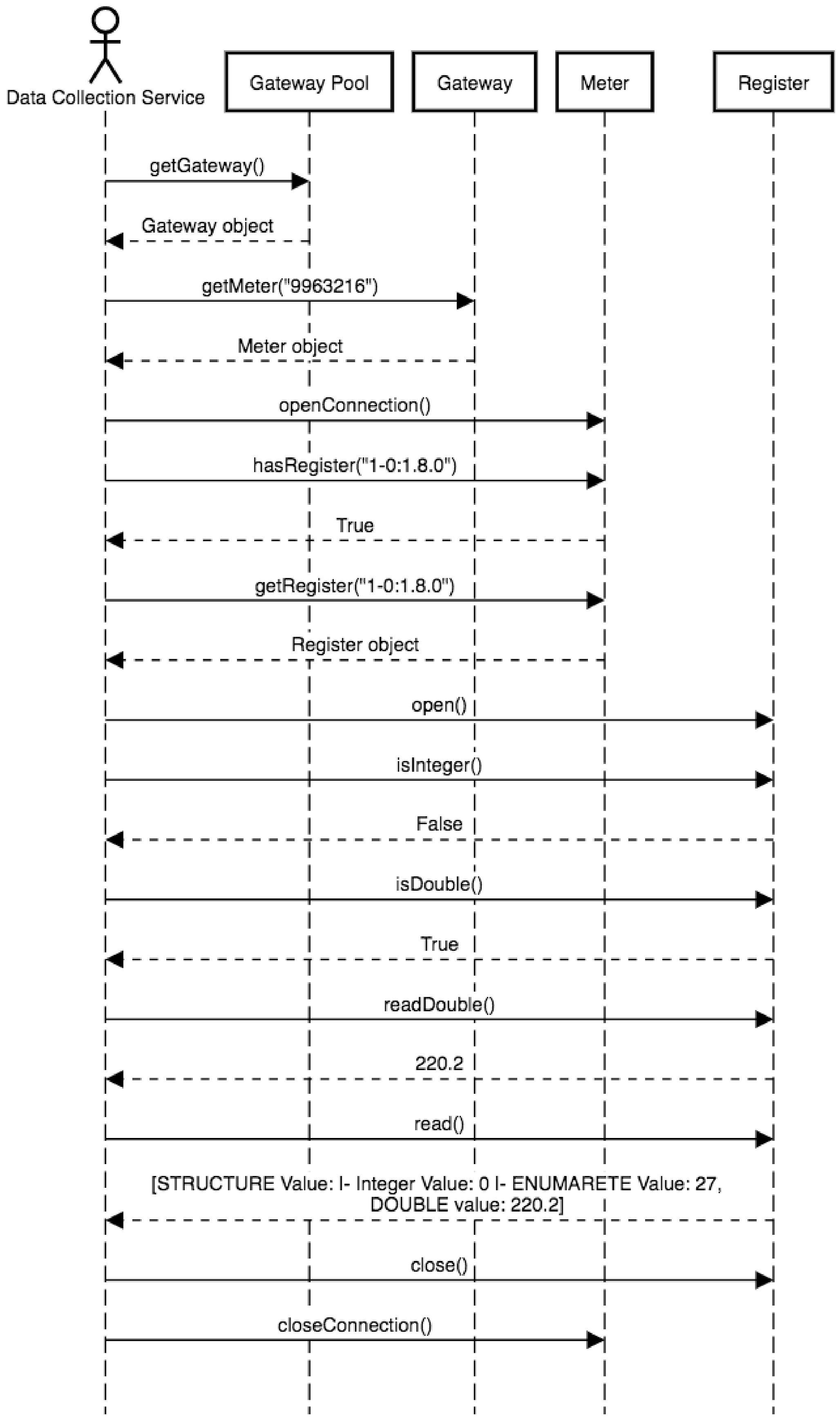


Figure 4.5: Sequence Diagram Describing Reading Registers

The data collection service then queries the meter by invoking the method hasRegister("1-0:1.8:0") on the meter object. The method hasRegister("1-0:1.8:0") checks the smart meter, if the register with the identifier "1-0:1.8:0" exist within the smart meter. The identifier for the register can be the number in case it is known for the developer what the number of the register is, or it can use the constants that are used in the generic API to easily identify the register used. The equivalent of using the register "1-0:1.8:0" as identifier is `POSITIVE_ACTIVE_ENERGY_TOTAL_ID`. However, the developer using the generic API has the freedom to use either the constant or the register number as an identifier. The method hasRegister("1-0:1.8:0") will return a boolean variable whether the register is found in the gateway or not. The meter object returns "True", which denotes that the meter has "1-0:1.8:0" register. The data collection service now is confident that the generic API has the register "1-0:1.8:0", so it can query the register and be sure that no exception of being not found will be thrown. Next the data collection service queries the register "1-0:1.8:0" by the method getRegister("1-0:1.8:0"), which returns the register object that has identifier "1-0:1.8:0". For the data collection service to be able to query the registers it has, it needs to invoke the open() function on the register, this allows all operations to be performed on the register. Then the data collection service can invoke functions that check the returned type of the register, in order to query the register with the appropriate method. Therefore, the data collection service first call the method isInteger() which checks if the measured read of the register is integer; the method returns False which means that the value is not integer. Then the data collection service calls function isDouble(); this returns True which indicates that the measured value according to the convention of the current smart meter is double. The data collection service can then be sure that the value within the register is double and can call the appropriate method which is readDouble(). In case the data collection service called the method readInteger() instead, it will not throw error. However, this will cast the double value to integer. In other words, calling method readInteger() will return 220 instead of 220.2 which shows that there is loss of accuracy, which is very crucial for smart meters. The data collection service then called the function read(), read() is the general method that is called no matter what type of the register it is. The read() function returns an array list of dataStore values which contains two elements, the first element is of type structures, that is an object that contains dataStore objects. This represent the meta data of the object as follows:

1. **Integer Value: 0:** A boolean value, representing if the register is writable or not, zero denotes the register is not writable.

2. **ENUMERATE Value: 27:** Represent the factor for the specific tariff that will be multiples by the measured number discussed in section.

The final value in the array list is the actual measurement of the register, which is 220.2 the same exact number as the returned value of readDouble(); however the object is encapsulated in dataStore object.

The final two methods called by the data collection service are the closing functions. First, the data collection service calls the method close() over the register object, in order to close the register object. Then the data collection service calls the method closeConnection() invoked on the meter object, which closes the connection between the gateway and the smart meter. Both functions do not return any object; however they are called in order to ensure the safety of the system to prevent accidental writes, also as the number of connections increase, this adds load to the gateway.

### 4.5.2 Writing Registers

Another sequence is described in Figure 4.6, where the data collection has the goal of writing or setting registers in the smart meter. The smart meter, in this example, is of type ANSI C12.18 that uses the OBIS code convention. The sequence diagram starts with the data collection service as the actor. To begin with, the data collection service queries the gateway pool by the method getGateway("200"); as a result the generic API will iterate over all gateways in the pool, but it will not find a gateway in our configuration file that has the identifier "200", as it does not exist in our system. Therefore, an exception of gateway "NoTFoundException" will thrown, exceptions are explained in more details in Chapter 5. The data collection service then call the method getGateway("2") and the gateway object is returned, as there exist in the gateway pool a gateway with identifier "2".

Next the returned gateway object is queried by the method getAllMetersByProtocol(VirtualProtocol) this method returns all meters that use the OBIS code convention. As mentioned before, the VirtualProtocol object indicates any protocol that use the OBIS code convention. The method will iterate over all meters connected to the gateway with identification "2" and will return all meters that use the OBIS code convention in a list of meter objects. The data collection service can iterate over all meters objects or perform get operation to return specific meter. In this sequence situation, the data collection service will just return the first meter.

Initiating the connection is the first step for the meter object, so we will a open connection on the selected meter from the meter list returned by the method getAllMetersByProtocol(VirtualProtocol) in order to start communicating with the smart meter. The openConnection() uses the parameters stored in the configuration file for initiating connection with the ANSI C12.18 smart meter. After the connection has been initiated, the data collection service calls hasRegister(`POSITIVE_ACTIVE_ENERGY_TOTAL_ID`) in order to check if the register with the identifier `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` exists in the smart meter. As shown in the sequence digram, the constant "`POSITIVE_ACTIVE_ENERGY_TOTAL_ID`" was used in this sequence situation, unlike the situation one, where the register used was the identifier by the OBIS itself. Using the constants is particularly useful because of two reasons. First, the developer does not have to memorize each and every register identification number he can just use the constant which is self explanatory. The second reason is that because of interoperability issues, different protocols have different naming convention. Therefore, the constants naming convention helps developers to unify the calling of registers. For example, a developer can just call `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` for any supported protocol in order to get the value of positive active energy total, such as in protocols like DLMS/COSEM and ANSI C12.18. The returned value of the function is True as the register with `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` identifier exists in the smart meter.

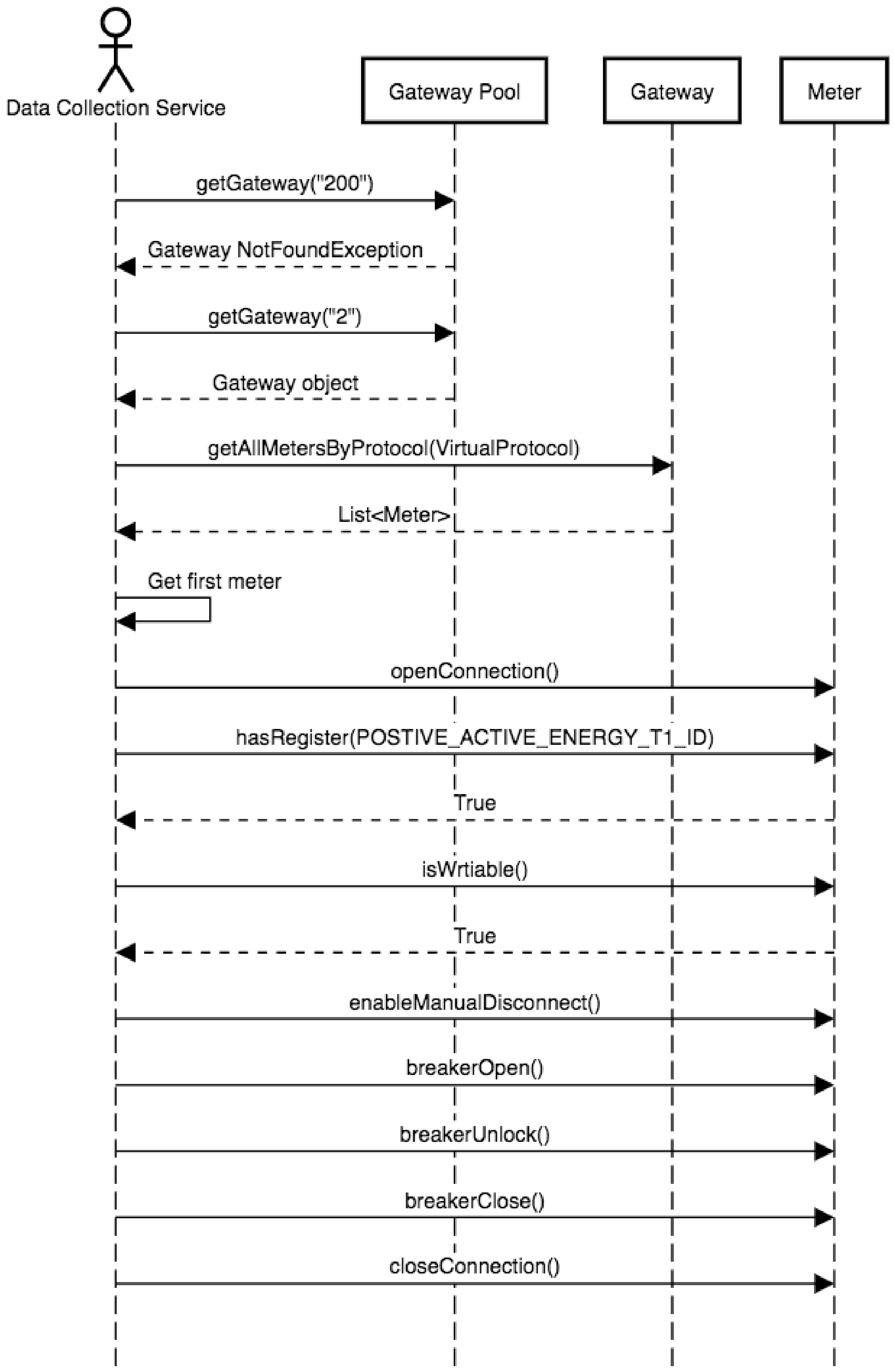


Figure 4.6: Sequence Diagram Describing Writing Registers

As the data collections service's goal is to set values in the smart meter, it first calls the method isWritable(), which checks if the smart meter selected allows writing or not. This is done by checking the configuration file writable field, which is either true or false variable for each smart meter. In case it allows writing this means that the meter object can invoke any write method without returning error.

This is a particularly useful method, because if the data collection service called any writing method, such as enableManuaDisconnect() and the meter does not allow writing, then an exception will be thrown that indicates that the meter does not allow write. Different types of exceptions are explained in more details in Chapter 5. For situation sequence two, the method isWritable() returns True, so that the data collection service can call writing methods to the meter object. The data collection service calls the method enableManualDisconnect(), which enables the user or the owner of the smart meter to turn it off. The ANSI C12.18 smart meter has a disconnect button which has the symbol [ 0 ]. When pressing this button and the manual disconnect register in the smart meter is activated, then the user can just disconnect the smart meter by pressing on the button. If the smart already enables the user to manually disconnect it, then nothing will happen if enablesManualDisconnect() is called, otherwise, the owner of the smart meter will be able to disconnect it. This is particularly important because the other write methods called by data collection service need the manual disconnect button to be activated. Next the data collection service call the method breakerOpen() which turns off the smart meter, so that electricity does not reach the electric appliance connected to the smart meter. It is important to mention that this state can also be reached by the owner of the smart meter when he presses the close button, but manual disconnect should be enabled. Then the data collection service calls the method breakerUnlock() which unlocks the meter, so that the owner of the meter can turn it on by pressing on the on button. However, in this situation the data collection service called the method breakerClose() which turned the meter back on without the owner of the meter pressing on the on button of the smart meter. The different states that the smart meter can have is presented in figure 4.4.3.12. Finally, the data collection service invoke the method closeConnection() on the meter object, in order to close connection that was opens by the data collection service. Closing the connection is done using the parameters that are recorded in the configuration file.

# Chapter 5

# Java Implementation

In this chapter we are going to discuss the Java binding to our generic API. In addition, we are going to explain in detail the implementation of the building blocks of our API, that were described in Chapter 4, of our API and explain details of some methods. Furthermore, we will explain in detail the exception handling of our system for different cases. Appendix A has the source code of the implementation in this chapter.

## 5.1 Classes Implementation

In this section we are going to discuss the different implementations of classes in our system concretely. We are going to discuss the implementation of each building block on its own, such as the gateway object, the meter object, the register object and the protocol object.

### 5.1.1 Gateway

The gateway implementation has two parts, the first part is GatewayFactory, which is responsible for getting the gateway object from the gateway pool by two methods getGateway(String gatewayID) and getGateway(), which returns gateway by identifier gatewayID or default gateway object respectively. The second part is related to the methods that are executed by the gateway object. All gateways objects implement the gateway interface. The implementation of the methods in the interface of gateway is developed by loading data related to meters connected to the gateways. This is done by private method metersFromConfigFile() which creates an array of meters that are connected to the gateway. The implementation of all of the methods use the array of meters to perform its function. For example, the method getAllMetersByProtocol(Protocol protocol) iterates through the array of meters, which is loaded from the configuration file and filters only meters that use the protocol "protocol". Another method that uses the array of meters is getAllSupportedProtocols() that loops over the array of meters and adds unique protocols to the array to be returned.

### 5.1.2 Meter

A meter object has supports those methods that were described in Chapter 4. As described in Chapter 4, the details about each meter connected to the gateway is recorded in the configuration file. The interface of the meter object is listed in Appendix A.1.1. Initially when the meter object is created, the it has parameters which are extracted from the configuration file such as:

- **MeterID:** Represents the serial number of the meter.
- **Protocol:** Represents the protocol used by the meter.
- **Manufacturer:** Represents the manufacturer of the meter.
- **Port:** Represents the port that is used for connection with the smart meter.
- **Location:** For ANSI C12.18 smart meter, the reading occurs by calling specific command line function, location represent the location of the executable.
- **Writable:** A boolean variable that represents whether the smart meter is writable or not.
- **Client Address:** Represents the client address that is used for the initial connection with the smart meter.
- **Logic Address:** Indicates the logic address of the meter in case the meter can measure more than one service such as electricity, water and gas.
- **Physical Address:** Indicates the address of the meter for initial connection with the smart meter acting as identifier for the connection.

As it is shown in Appendix A.1.2, the MeterFactory has two constructor, one to handle the initialization of the DLMS/COSEM protocol smart meters and the other to handle the initialization of ANSI C12.18 protocol smart meters. The reason of separating the initialization of each protocol smart meters is that the parameters that needed to open connection between gateway and DLMS/COSEM smart meter is different than the parameters that are needed to open connection between gateway and ANSI C12.18 smart meter. Several methods in the MeterFactory are implemented internally in two ways, one to support the DLMS/COSEM protocol and the other to support the ANSI C12.18 protocol. For example, the openConnection() method listed in Appendix A.1.3 is divided into two parts depending on the protocol used. If the method is invoked over a DLMS/COSEM smart meter, then the connection is initialized using the method and the parameters for DLMS/COSEM connection, otherwise, if the method is invoked over a ANSI C12.18, smart meter then the internal part related to the implementation of connecting to an ANSI C12.18 smart meter is executed. This way the method openConnection() could be used for both DLMS/COSEM smart meters and ANSI C12.18 smart meters without the developer caring about the underlying protocol connection implementation. In addition, there exist some methods that have only one internal implementation for both protocols such as getRegister(String registerID) that returns specific register by its identifier, and getManufacturer() that returns the manufacturer of the smart meter, described at Appendix A.1.4 and A.1.4.1 respectively.

The write methods described in Chapter 4 is invoked over the meter object. However, the smart meter used in this project a DLMS/COSEM meter and an ANSI C12.18 meter. The DLMS/COSEM smart meter does not allow writing, on the other hand the ANSI C12.18 meter allows writing. Methods like breakerClose() and enableManualDisconnect() listed in Appendix A.1.5 and A.1.6 allow an ANSI C12.18 meter to write but if it is invoked over a DLMS/COSEM smart meter, an exception NotWritableMeterException will be thrown. Writing methods that are invoked on the ANSI C12.18 smart meter call batch files such as BreakerClose.bat and EnableManualDisconnect.bat in order to close the breaker or enable manual disconnect respectively.

### 5.1.3 Register

As smart meters have registers, the meter object is queried to return specific register or all registers within the smart meter. The interface implemented object is listed in Appendix A.2.1. RegisterFactory implements the interface mentioned at Appendix A.2.1. One of the most crucial methods that can be invoked on the register object is read() method. The method read() returns an ArrayList of DataStore objects that contains meta data about the register and the measured values. The method read() is implemented in registerFactory is listed in Appendix A.2.2. The method implementation shows the call to the bash command line script in case the meter that invokes the method on the register is ANSI C12.18; otherwise the method uses the JDLMS library, which is discussed in Subsection 5.2.1, in order to read data from the DLMS/COSEM smart meter which returns an attributesList, which is an ArrayList of dataStore objects. The variable clientDLMS that is an instance variable of register objects indicates whether the register is inside a DLMS/COSEM or an ANSI C12.18 smart meter. It is a boolean variable because a smart meter can be either of type DLMS/COSEM or ANSI C12.18 smart meter but not both.

Another set of methods supported by the register object is reading the object by its value. These are extremely useful methods in case the developer is sure the type of register used. Because, unlike dataStore object, these methods do not read the value of the measured value and encapsulate it with its type and meta data. These methods just return the measured value with the expected type. If the method that is invoked on the register object reads a different type of object, either there will be a loss of data or an exception will be thrown. These methods are listed bellow:

- readDouble()
- readInteger()
- readLongInteger()
- readBoolean()

- readString()

For example, the method readDouble() is implemented to return the double value that is measured by the smart meter stored in the register object. The method is listed in Appendix A.2.3. The method has two internal implementations, one for DLMS/COSEM reads, while the other one is for the ANSI C12.18 reads. The DLMS/COSEM part uses the dataStore object described before, to extract the measured value and cast it to double. In the case of ANSI C12.18, the part of the method related to reading calls the bash command line script in order to read the measurements; then the method calls Thread.sleep(18000) in order to sleep the main thread. This is done because reading of register data takes time, so if the program continues and reads the register value, it will not be updated. In other words, sleeping the thread give time for the smart meter to update its registers. After the thread is woken up, the method checks if the value of the register is double; if the register type is not double, then an exception will be thrown for having an invalid type of the register. The methods readInteger(), readLongInteger(), readBoolean() and readString() function the same way as readDouble().

Another two classes that were described in Chapter 4 are registerType and registerUnit. The implementation of both classes is an enum that has all different possible values that can be assigned to the type of register and the unit of register, for example the possible units for register measured values are Kw, W, kWh, kvarh, Wh and others. The classes registerType and registerUnit are listed in Appendices A.2.4 and A.2.5 respectively.

### 5.1.4 Protocol

The protocol type is implemented in our system as an interface; every protocol that is supported by our generic API should have a class that implements this protocol interface. The protocol interface contains only one method that is getName() which return the name of each protocol as a string. As mentioned before, in our experiment, we have used two different smart meters that use two different protocols namely DLMS/COSEM and ANSI C12.18. The DLMS/COSEM protocol is implemented as an interface named DLMSProtocol that implements the protocol interface; the DLMSProtocol interface contains all necessary information related to register identifiers, with their name, as well as the types and units of the register. The DLMSProtocol implementation is listed in Appendix A.2.6; however only the data of one register are listed, as listing the entire registers identifier with register type and unit will be very long. The data related to registers such as register identifiers, units and types was done by iterating on the COSEM objects in the smart meter using the JDLMS library that is described at section 5.2.1. The second protocol that is supported by our generic API is ANSI C12.18. The ANSI C12.18 supports the OBIS code convention. Therefore, we implemented interface VirtualProtocol that extends the protocol interface, which supports any protocol that uses the OBIS code convention. The interface contains the register identifiers, unit and type. The OBIS code convention helps VirtualProtocol to support different protocols without relaying on the underlying specification of the protocols. The interface class of VirtualProtocol is listed in Appendix A.2.7; however not all registers are shown as it is a very long list of registers. Even though we are separating the implementation of both protocols, the developer can call a register by its constant name such as `POSITIVE_ACTIVE_ENERGY_TOTAL_ID` which will return the positive active energy total register identifier for both protocols, so that the developer does not care about the underlying protocol.

## 5.2 Connection with Smart Meters

In the project we were using two types of smart meters. DLMS/COSEM and ANSI C12.18. In this section we will describe the method of connecting the smart meters with the gateway and the libraries used to develop the implementation in Java.

### 5.2.1 JDLMS

The library JDLMS [26] is a Java library for DLMS/COSEM protocol. JDLMS was used in the project to initiate a connection, exchange data between gateway and DLMS/COSEM smart meter and close connection. The JDLMS library contains two stacks client stack and server stack. A client stack is used to access data within the smart meter, such as registers over TCP/IP or serial communication. The server stack allows the building of a DLMS/COSEM server which can be appropriately customized. In this project we used JDLMS for accessing data in the meter using the client stack over serial connection. The JDLMS library client stack supports the following features.

- **Transport:** JDLMS support variety of methods for transport. In this project we used HDLC (IEC 62056-46) via serial RS-485.

- **Object addressing:** JDLMS support logic naming referencing and short referencing. However in this project we only used short naming referencing as it was only supported by the smart meter used.

- **Authentication mechanisms:** JDLMS allow several mechanisms for authentication. However in this project we used None which indicates no authentication was performed.

- **Encryption mechanism:** JDLMS support AES-GCM-128 specified in RFC 3394. However in this project we did not use the encryption mechanism

- **Data transfer service:** JDLMS supports get, set and action. In this project we used the get data transfer service to obtain the data at register.

As mentioned before, our generic API uses the JDLMS library in order to initiates connection, exchange data and close the connection. In order to initiate the connection we defined the class JdlmsClient, The class JdlmSClient has a constructor that initializes the connection with the DLMS/COSEM smart meter connected. The constructor for the JdlmsClient class is listed in Appendix A.2.10. The constructor takes the following arguments:

- **UseSerialPort:** UsesSerialPort is a boolean that value indicates that the connection use serial RS-485.
- **SerialPort:** SerialPort is a string that denotes the serialPort used for the connection.
- **Ip:** Ip represents the IP address as a string in case the UsesSerialPort is false, which means that the connection with the meter is TCP/IP not serial RS-485.
- **Port:** The port number indicates the port used when using TCP/IP connection.
- **ClientAdress:** ClientAddress is an integer which denotes the client address needed in the connection. The client address value for each meter should be recorded in the configuration file to enable the serial connection.
- **Password:** In case the smart meter needs a password for reading or writing the data, SecuritySuiteBuilder is used which is a method created by the JDLMS library for authentication. In case there is no password then the authentication will not happen and the default security access level will be imposed.
- **LogicalDeviceAddress:** Denotes the logical device address which is recorded in the configuration file per meter.
- **PhysicalDeviceAddress:** Denotes the physical device address which is recorded in the configuration file per meter.
- **IntialzeSnObjects:** This is a boolean variable that indicates if there should be a mapping between the register number and the value it currently has. This is particularly useful when we want to initiate a connection and read the values in the registers and save them in the map object.

The constructor initiates a connection by a connectionBuilder object, which is defined by the JDLMS library in order to open a connection with the DLMS/COSEM smart meter. The object connectionBuilder has methods that append information regarding logical device address, physical device address, client ID and referencing method. Having these parameters from the constructor enables JDLMS to initiate the connection.

### 5.2.2 Bash Command Line Script for ANSI C12.18 Protocol

In the previous section we discussed the JDLMS library which enables us to initiate a connection and exchange data with DLMS/COSEM smart meters. In this section we will discuss how we were able to initiate the connection and exchange data between the gateway and the ANSI C12.18 smart meters. In order to read the register readings we run a bash script. Bash is a command language interpreter; we use it for reading because binary executable needs to be run that is responsible for communicating with the smart meter. CmdMeterCommand.exe is the executable that is able to communicate with the ANSI C12.18 smart meters. The readBillingData.bat is listed Appendix A.2.11, which is the batch script that is responsible for reading the current values of all registers and write them out to an XML file with the register identifier and current value of registers.

## 5.3 Exception Handling

In this section we will explain in detail the implementation of different exceptions that are handled by our generic API. As discussed in the previous sections, there are several unexpected operations that can cause exceptions to occur in the system. The generic API defines several check exceptions, which are used to explicitly show the user of our generic API that the code may cause an error, so developers needs to be careful when using the method [23]. In addition, the check exception show the user what kind of exception or error may occur. Those exceptions need to be handled by the user of the generic API [23].

Our generic API implements several exception classes that extends the class Throwable which is the superclass of all errors and exceptions in the Java language. Each exception class handles specific errors that may happen while the user of the generic API is using the API. The classes GatewayNotFoundException and MeterNotFoundException are listed in Appendices A.2.8 and A.2.9 respectively. The exception classes supported by our generic API are listed bellow:

- **GatewayNotFoundException:** GatewayNotFoundException is thrown when getGateway(String gatewayID) is called and the gateway pool does not contain a gateway whose identifier equals the gatewayID parameter in the method.

- **MeterNotFoundException:** MeterNotFoundException is thrown when trying to get or access a meter that is not connected to the gateway that the method is invoked on. For example, the exception is thrown, if we run method getMeter(String meterID) and there does not exist a meter in the configuration file with identifier equals the meterID parameter in the method.

- **ConnectionException:** ConnectionException happens when there exists an error in the connection with the smart meter. For example, the exception is thrown when we run the method openConnection() on a meter object and the invoked meter connection has an error such as when connecting over serial RS-485 and the physical connection does not exist.

- **InValidConfigFileException:** InValidConfigFileException is thrown when the data in the configuration file are not recorded appropriately according to the convention (as a CSV file where data are organized in the conventional order specified in Figure 4.3).

- **InValidDataTypeException:** InValidDataTypeException is thrown when the user calls a read method that reads register element based on its type such as readDouble() and the register type is not double. Therefore, an exception should be thrown as there is an inconsistency in the data types. The user of such methods like readDouble() should be sure that the type of the register is double, before calling the method.

- **NotWritableMeterException:** The exception NotWritableMeterException is thrown when trying to use write methods over a non writable smart meter.

- **RegisterNotFoundException:** RegisterNotFoundException is thrown when the user tries to get or access a register with a wrong naming convention, or the register does not exist within the smart meter.

## 5.4 Generic API Demonstration

In this section we will describe Java programs that use our developed generic API discussed above. We demonstrate the methods with both protocols and perform reading and writing operations. We will start by describing a situation using the DLMS/COSEM smart meter, then we will describe a situation using ANSI C12.18 smart meter.

### 5.4.1 DLMS/COSEM Demonstration

In this subsection we will consider that the gateway has a DLMS/COSEM smart meter that is connected to it. The method described in Listing 5.1, shows a test method namely testDLMS(), for the API which use DLMS/COSEM protocol. The method begins by getting the default gateway from the gateway pool. Then using the return gateway it calls method getMeter("96224681") which returns the meter with the identifier 96224681. Using the configuration file the generic API knows automatically that the meter with identifier 96224681 uses the DLMS/COSEM protocol. After that it opens the connection using method openConnection() which initiates the connection using JDLMS library to start data exchange. Then we query the meter to check if a register with the identification number "1.1.16.7.0.255" exists in the meter. The meter object returns true as it exists within the meter. Then we check the type of the register with getRegisterType() which returns a double. The next step is to open the register with the method open. We can then run the method read() and it will not throw exception because we are sure that the register with the identifier exists in the system. We call the same read() method but with `"INSTATANEOUS_POWER_ID"` as identifier and it returns the same value. The reason is that the DLMSProtocol has the predefined values of registers, so it maps the identifier "1.1.16.7.0.255" to `"INSTATANEOUS_POWER_ID"`. Both methods will return an ArrayList of DataStore values which represent the measured values and types as well as meta data. Moreover, we call method readDouble() that returns the value of the meter as a double. We are sure that InValidDataTypeException will not be thrown, because we are sure that the type of the data in register is double.

```java
public static void testDLMS() throws GatewayNotFoundException, Exception,
    ↪ InValidConfigFileException,
   MeterNotFoundException, ConnectionException, InValidDataTypeException {
  Gateway gateway = GatewayFactory.getGateway();
  Meter meter = gateway.getMeter("96224681");
  meter.openConnection();
  System.out.println(meter.hasRegister("1.1.16.7.0.255"));
  System.out.println(meter.getRegister("1.1.16.7.0.255").getRegisterType());
  meter.getRegister("1.1.16.7.0.255").open();
  System.out.println(meter.getRegister("1.1.16.7.0.255").read());
  System.out.println(meter.getRegister(DLMSProtocol.INSTANTANEOUS_POWER_ID).read()
      ↪ );
  System.out.println(meter.getRegister("1.1.16.7.0.255").readDouble());
  meter.getRegister("1.1.16.7.0.255").close();
  meter.closeConnection();
 }
```

Listing 5.1: testDLMS() Method Implementation

### 5.4.2 ANSI C12.18 Demonstration

In this subsection we will consider that the default gateway has an ANSI C12.18 smart meter that is connected to it. The method described in Listing 5.2 shows a test method namely testANSI() for the API which uses the ANSI C12.18 protocol. The method starts with getting the default gateway from the gateway pool as we did in Subsection 5.4.1. We invoke the method getMeter("UB126001034") on the default gateway object where the parameter "UB126001034" defines the serial number of the ANSI C12.18 meter. The generic API will recognizes the smart meter with identifier "UB126001034" as ANSI C12.18 smart meter, as it is recorded in the configuration file. The connection will be opened with the smart meter using openConnection() that will use the bash script to start the connection with the smart meter. Then we will get the first register in the smart meter using the method getAllRegisters() that returns the list of all registers in the order of there occurrences in the OBIS code. We will then print the read value of the register knowing that it has the type double. Moreover, We will get the register with the identification `POSITIVE_ACTIVE_ENERGY_TOTAL_ID`, to get the register that measures the positive active energy total. We invoke the method read on the register object which returns an ArrayList of DataStore objects which contain the measured value of the register as well as the type of the register. The next method invoked on the register object is getRegisterUnit() that returns the unit watt. Then we call getRegisterID() and getMeter().getMeterID() to get the identifier of both the register and of the smart meter that contains the register respectively. In addition, for the ANSI C12.18 we can perform write operation, therefore we check at the beginning if the meter is writable by invoking the method isWritable() on the meter object. It returns true, so we can call a write method such as breakerClose(). If the meter was not writable, we would have received NotWritableMeterException, that is the reason why we did the isWritable() check first. Another register that we query the meter about is "pulseInputChannelOneTotal", that returns register. We used the returned register object to get its type. The returned type of the register is none because the "pulseInputChannelOneTotal" does not have specific type. Finally we iterate over all registers in the meter and we outputted each register identifier followed by the its type. The final two invoked methods close any open register and close the connection at the end.

```java
public static void testANSI() throws GatewayNotFoundException, Exception,
    ↪ InValidConfigFileException,
   MeterNotFoundException, NotWritableMeterException, InValidDataTypeException,
      ↪ ConnectionException {
  Gateway gateway = GatewayFactory.getGateway();
  Meter meter = gateway.getMeter("UBI26001034");
  meter.openConnection();
  meter.getAllRegisters().get(0).open();
 System.out.println(meter.getAllRegisters().get(0).readDouble());
 Regsiter register = meter.getRegister(VirtualProtocol.
    ↪ POSITIVE_ACTIVE_ENERGY_TOTAL_ID).open();
 System.out.println(register.read());
 System.out.println(register.getRegisterUnit()); //W
 System.out.println(register.getRegisterID()); //"1-0:1.8.0"
 System.out.println(register.getMeter().getMeterID()); //"UBI26001034"

 if (meter.isWritable()) {
  meter.breakerClose(); // close breaker
 }
 System.out.println(meter.getRegister("pulseInputChannelOneTotal").getRegisterType
    ↪ ());//RegisterUnit.none

  // printing register id and register type
 for (int i = 0; i < meter.getAllRegisters().size(); i++) {
  System.out.println(meter.getAllRegisters().get(i).getRegisterID() + "␣" + meter.
     ↪ getAllRegisters().get(i).getRegisterType());
 }

  meter.getAllRegisters().get(0).close();
  meter.closeConnection();
 }
```

Listing 5.2: testANSI() Method Implementation

# Chapter 6

# Conclusion

## 6.1 Summary

The main focus of the current thesis was developing a generic API that can interpret the features of protocols, which are used by different smart meters and IoT applications, in a common API that can be used to communicate directly with a cloud provider or the data collection service without knowing the underlying implementation details. The first part of the thesis included researching and documenting the state-of-the-art in the area of smart meters, IoT application and how gateways can be used to overcome the interoperability issues. Interoperability is the ability for a system to exchange data with other system and the interface for communication is understood by both sides. The interoperability issues that face smart meters and IoT applications are different underlaying protocols used for communication, different naming conventions, different data types of information and different units for recording the data. In addition we have documented and researched the specification of each of the protocols used in the project namely DLMS/COSEM and ANSI C12.18. Then we were able to design and implement the generic API that developers can use in order to exchange data with any of the smart meters with a general approach without developer thinking about what is the protocol used by the smart meter.

In this project we were able to develop a generic API that overcomes the interoperability challenge in smart meters and IoT applications in general. We were able to describe the API in abstract way, also we gave several sequences situation for using the API. After describing the API in abstract way we described the binding of our API to Java programming language. By describing the Java binding we explained the building blocks of the API using Java language. The main building blocks for our generic API were Gateway, Meter, Register and protocol. We were able to develop other objects to overcome the interoperability challenge such as registerUnit, registerType, dataStore objects. The result was an implementation of the specification of each protocol in the DLMSProtocol for the DLMS/COSEM smart meter and a VirtualProtocol that covers any meter that used OBIS code which supports ANSI C12.18.

## 6.2 Future Work

The solution and the research work started in this master thesis can be further on continued in several directions. To begin with, as our generic API was tested only with smart meters, the generic API can be extended with the same concepts in order to support other IoT devices. There are wide varieties of IoT devices that the generic API can be extended to support. For example, machines in factory can have an "on field" gateway device where the generic API converts the API using machines communication protocols such as the MQTT protocol. Therefore another protocol that can be easily supported by our system in the future is the MQTT protocol, which is widely used in many applications.

One of the challenges we faced in this project is that the performance of the API was not good in terms of reading. For example for the ANSI C12.18 smart meter we need to sleep the main thread in order to wait for the new value read which is not optimum. Therefore, another future direction of research can be headed towards the development of new algorithms that ensure not only optimum time and memory complexity of the generic API; as these algorithms will be used by developers, these must at least offer the same time complexity as working with the protocol natively, i.e. as if working without the generic API.

The subject of the interoperability challenge in IoT application and specifically smart meters is currently a hot topic, as the number of IoT devices have increased drastically and also smart meters are more and more used world wide. Therefore, the topic of how to address the arising interoperability challenges remains an open area of research.

# Appendices

# Appendix A

# Java Classes

In this appendix we will show the implementation of different Java classes and methods used by our generic API system.

## A.1 Meter Class Implementation

### A.1.1 Meter Interface

```java
package meter;

import java.io.IOException;
import java.util.ArrayList;

import exceptions.NotWritableMeterException;
import exceptions.RegisterNotFoundException;
import protocol.Protocol;
import register.Register;

public interface Meter {

  public ArrayList<Register> getAllRegisters();

  public String getManufacturer();

  public Protocol getProtocol();

  public Register getRegister(String registerID) throws RegisterNotFoundException;

  public String getMeterID();

  public boolean hasRegister(String registerID);

  public void openConnection() throws Exception;

  public void closeConnection() throws IOException;

  public boolean isWritable();

  public void breakerClose() throws IOException, NotWritableMeterException;

  public void breakerOpen() throws IOException, NotWritableMeterException;

  public void breakerUnlock() throws IOException, NotWritableMeterException;

  public void disableManualDisconnect() throws IOException,
      ↪ NotWritableMeterException;

  public void enableManualDisconnect() throws IOException,
      ↪ NotWritableMeterException;

}
```

### A.1.2 MeterFactory Constructor

```java
// several constructors for different protocols
  public MeterFactory(String protocol, String meterID, String manufacurer, String
      ↪ port, String clientAdress,
      String logicAdress, String physicalAdress, String writable) throws Exception
          ↪ {
    this.manufacurer = manufacurer;
    if (protocol.equals("DLMSCOSEM")) {
      this.protocol = new DLMSProtocolFactory();
    }

    this.meterID = meterID;
    this.clientAdress = Integer.parseInt(clientAdress);
    this.logicAdress = Integer.parseInt(logicAdress);
    this.physicalAdress = Integer.parseInt(physicalAdress.charAt(0) + "");
    this.port = port;
    this.writable = writable.equals("true") ? true : false;
  }

  // constructor for ansi smart meters
  public MeterFactory(String protocol, String meterID, String manufacurer, String
      ↪ port, String location,
      String writable) throws IOException, IllegalArgumentException,
          ↪ IllegalAccessException {
    this.meterID = meterID;
    this.writable = writable.contains("true") ? true : false;
    this.manufacurer = manufacurer;
    this.port = port;
    this.protocol = new VirtualProtocolFactory(protocol);
    this.location = location;
  }
```

### A.1.3 openConnection() Method Implementation

```
// DLMS/COSEM => opening connection and reading the registers definitions
  public void openConnection() throws Exception {
    if (protocol instanceof DLMSProtocol) {
      client = new JDlmsSampleClient(true, port, null, null, clientAdress, null,
          ↪ logicAdress, physicalAdress,
          false);
      for (String shortName : client.getShortNames()) {
        Meter m = new MeterFactory("DLMS/COSEM", meterID, manufacurer, port,
            ↪ clientAdress + "",
            logicAdress + "", physicalAdress + "", writable + "");
        registersList.add(new RegisterFactory(shortName, m, client));
      }
    }

    // if protocol is virtualProtocol then it use OBIS and will have the
    // registerName from xml
    if (protocol instanceof VirtualProtocol) {
      File file = new File(location + "/results/result4.xml");
      DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.
          ↪ newInstance();
      DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder()
          ↪ ;
      Document document = (Document) documentBuilder.parse(file);
      NodeList nList = document.getElementsByTagName("measurement");

      for (int i = 0; i < nList.getLength(); i++) {
```

```
      if (!nList.item(i).getParentNode().getParentNode().getNodeName().equals("adhocRead")) {
        break;
      }
      Node n = nList.item(i);
      Element e = (Element) n;
      Meter m = new MeterFactory("ANSI", meterID, manufacurer, port, location, writable + "");
      registersList
          .add(new RegisterFactory(e.getAttribute("registerName"), m, location, e.getAttribute("unit")));
    }

  }

}
```

### A.1.4 getManufacturer() Method Implementation

```
@Override
  public String getManufacturer() {
    return manufacurer;
  }
```

#### A.1.4.1 getRegister(String registerID) Method Implementation

```
@Override
  public Register getRegister(String registerID) throws RegisterNotFoundException {
    for (Register register : registersList) {
      if (register.getRegisterID().equals(registerID)) {
        return register;
      }
    }
    throw new RegisterNotFoundException("RegisterID␣is␣not␣found");
  }
```

### A.1.5 breakerClose()) Method Implementation

```java
@Override
  public void breakerClose() throws IOException, NotWritableMeterException {
    if (!writable) {
      throw new NotWritableMeterException("Meter␣doesn’t␣authorize␣write");
    }
    Runtime.getRuntime().exec("cmd␣/c␣start␣\"\"␣" + location + "/BreakerClose.bat"
        ↪ );

  }
```

### A.1.6 enableManualDisconnect() Method Implementation

```java
@Override
  public void enableManualDisconnect() throws IOException,
      ↪ NotWritableMeterException {
    if (!writable) {
      throw new NotWritableMeterException("Meter␣doesn’t␣authorize␣write");
    }
    Runtime.getRuntime().exec("cmd␣/c␣start␣\"\"␣" + location + "/
        ↪ EnableManualDisconnect.bat");

  }
```

## A.2 Register Class Implementation

### A.2.1 Register Interface

```java
package register;

import java.io.IOException;
import java.util.ArrayList;

import javax.xml.parsers.ParserConfigurationException;

import org.openmuc.jdlms.datatypes.DataObject;
import org.xml.sax.SAXException;

import exceptions.ConnectionException;
import exceptions.InValidDataTypeException;
import meter.Meter;

public interface Register {

  public String getRegisterID();

  public RegisterType getRegisterType() throws IOException;

  public RegisterUnit getRegisterUnit();

  public void open() throws ConnectionException, IOException, InterruptedException
      ↪ ;

  public void close() throws ConnectionException;

  // changed register value to dataObject to make it retrun dataObject which is
      ↪ general of DLMS/COSEM
  public ArrayList<DataObject> read() throws ConnectionException, IOException,
      ↪ ParserConfigurationException, SAXException, InterruptedException;

  public Meter getMeter();

  public double timeReads() throws IOException;

  public double readDouble() throws IOException, InValidDataTypeException,
      ↪ InterruptedException;

  public Integer readInteger() throws IOException, InValidDataTypeException,
      ↪ ParserConfigurationException, SAXException, InterruptedException;

  public boolean readBoolean() throws IOException, InValidDataTypeException,
      ↪ InterruptedException;

  public long readLongInteger() throws IOException, InValidDataTypeException,
      ↪ InterruptedException;

  public String readString() throws IOException, InValidDataTypeException;



}
```

### A.2.2 read() Method Implementation

```
// returns arrayList of dataObject, in case of DLMS/COSEM could be structure
 // In case of Ansi returns one value which is the value of the read in
 @Override
 public ArrayList<DataStore> read()
     throws ConnectionException, IOException, ParserConfigurationException,
         ↪ SAXException, InterruptedException {
   if (clientDLMS != null) {
     return attributesList;
   }
   Thread.sleep(18000);
   Runtime.getRuntime().exec("cmd␣/c␣start␣\"\"␣" + location + "/ReadBillingData.
       ↪ bat");
   ArrayList<DataStore> ansiValue = new ArrayList<DataStore>();
   ansiValue.add(getRegisterValueANSI());
```

### A.2.3 readDouble() Method Implementation

```
 @Override
 public double readDouble() throws IOException, InValidDataTypeException,
     ↪ InterruptedException {
   if (clientDLMS != null) {
     for (DataObject attribute : attributesList) {
       if (attribute.getType().toString().contains("DOUBLE")) {
         return new Double(attribute.getRawValue().toString());
       }
     }
   } else {
     Runtime.getRuntime().exec("cmd␣/c␣start␣\"\"␣" + location + "/ReadBillingData
         ↪ .bat");
     Thread.sleep(18000);
     try {
       if (this.getRegisterType().toString().equals("Double")) {
         return new Double((Double) getRegisterValueANSI().getRawValue());
       }
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
   throw new InValidDataTypeException("The␣register␣doesn't␣contain␣double␣value")
       ↪ ;
```

```
  }
```

### A.2.4 RegisterType Class Implementation

```
package register;

/*
 * Type returned by the registers
 *
 */
public enum RegisterType {
  String, Integer, Double, Boolean, Long;
}
```

### A.2.5 RegisterUnit Class Implementation

```
package register;

/*
 * Possible unites returned by registers;
 */
public enum RegisterUnit {

  Kw, W, kWh, kvarh, Wh, varh, none, seconds;
}
```

### A.2.6 DLMSProtocol Interface Implementation

```
package protocol;

import register.RegisterType;
import register.RegisterUnit;

public interface DLMSProtocol extends Protocol {
  // constants related to DLMS/COSEM that doesn't follow the OBIS code

  public static final String INSTANTANEOUS_POWER_ID = "1.1.16.7.0.255";
  public static final RegisterUnit INSTANTANEOUS_POWER_UNIT = RegisterUnit.W;
```

```java
  public static final RegisterType INSTANTANEOUS_POWER_TYPE = RegisterType.Integer
      ↪ ;

}
```

### A.2.7 VirtualProtocol Interface Implementation

```java
package protocol;

import register.RegisterType;
import register.RegisterUnit;

// implementation of general protocol handling
public interface VirtualProtocol extends Protocol {

  // OBIS standard constants
  public static final String POSITIVE_ACTIVE_ENERGY_TOTAL_ID = "1-0:1.8.0";
  public static final RegisterUnit POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT =
      ↪ RegisterUnit.kWh;
  public static final RegisterType POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE =
      ↪ RegisterType.Double;

  public static final String POSITIVE_ACTIVE_ENERGY_TOTAL_ID_T1 = "1-0:1.8.1";
  public static final RegisterUnit POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT_T1 =
      ↪ RegisterUnit.kWh;
  public static final RegisterType POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE_T1 =
      ↪ RegisterType.Double;

  public static final String POSITIVE_ACTIVE_ENERGY_TOTAL_ID_T2 = "1-0:1.8.2";
  public static final RegisterUnit POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT_T2 =
      ↪ RegisterUnit.kWh;
  public static final RegisterType POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE_T2 =
      ↪ RegisterType.Double;

  public static final String POSITIVE_ACTIVE_ENERGY_TOTAL_ID_T3 = "1-0:1.8.3";
  public static final RegisterUnit POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT_T3 =
      ↪ RegisterUnit.kWh;
  public static final RegisterType POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE_T3 =
      ↪ RegisterType.Double;

  public static final String POSITIVE_ACTIVE_ENERGY_TOTAL_ID_T4 = "1-0:1.8.4";
  public static final RegisterUnit POSITIVE_ACTIVE_ENERGY_TOTAL_UNIT_T4 =
      ↪ RegisterUnit.kWh;
  public static final RegisterType POSITIVE_ACTIVE_ENERGY_TOTAL_TYPE_T4 =
      ↪ RegisterType.Double;

  public static final String NEGETIVE_ACTIVE_ENERGY_TOTAL_ID = "1-0:2.8.0";
  public static final RegisterUnit NEGETIVE_ACTIVE_ENERGY_TOTAL_ID_UNIT =
      ↪ RegisterUnit.kvarh;
  public static final RegisterType NEGETIVE_ACTIVE_ENERGY_TOTAL_ID_TYPE =
      ↪ RegisterType.Integer;

  public static final String ABSOLUTE_ACTIVE_ENERGY_TOTAL_ID = "1-0:15.8.0";
  public static final RegisterUnit ABSOLUTE_ACTIVE_ENERGY_TOTAL_UNIT =
      ↪ RegisterUnit.kWh;
  public static final RegisterType ABSOLUTE_ACTIVE_ENERGY_TOTAL_TYPE =
      ↪ RegisterType.Double;

  public static final String POSITIVE_REACTIVE_ENERGY_TOTAL_ID = "1-0:3.8.0";
  public static final RegisterUnit POSITIVE_REACTIVE_ENERGY_TOTAL_UNIT =
      ↪ RegisterUnit.kvarh;
  public static final RegisterType POSITIVE_REACTIVE_ENERGY_TOTAL_TYPE =
      ↪ RegisterType.Double;


  public static final String POWER_OUTAGE_DURATION_ID = "powerOutageDurationTotal"
      ↪ ;
  public static final RegisterUnit POWER_OUTAGE_DURATION_UNIT = RegisterUnit.
      ↪ seconds;
  public static final RegisterType POWER_OUTAGE_DURATION_TYPE = RegisterType.Long;


  public static final String PULSE_INPUT_CHANNEL_ONE_TOTAL_ID = "
      ↪ pulseInputChannelOneTotal";
  public static final RegisterUnit PULSE_INPUT_CHANNEL_ONE_TOTAL_UNIT =
      ↪ RegisterUnit.none;
  public static final RegisterType PULSE_INPUT_CHANNEL_ONE_TOTAL_TYPE =
      ↪ RegisterType.Boolean;
}
```

### A.2.8 GatewayNotFoundException Class Implementation

```
package exceptions;
```

```java

public class GatewayNotFoundException extends Throwable {

  public GatewayNotFoundException(Exception e) {
    super(e.getMessage());
  }
}
```

### A.2.9 MeterNotFoundException Class Implementation

```java
package exceptions;

public class MeterNotFoundException extends Throwable {

  public MeterNotFoundException(String s) {
    super(s);
  }
}
```

### A.2.10 JDLMSClient Constructor Implementation

```java
public JdlmsClient(boolean useSerialPort, String serialPort, String ip, Integer
    ↪ port, Integer clientAddress,
   String password, Integer logicalDeviceAddress, Integer physicalDeviceAddress,
      ↪ boolean initializeSnObjects)
   throws Exception {
  this.initializeSnObjects = initializeSnObjects;
  if (useSerialPort) {
   SerialConnectionBuilder serialConnectionBuilder = new SerialConnectionBuilder(
      ↪ serialPort);
   connectionBuilder = serialConnectionBuilder;
   serialConnectionBuilder.setBaudRateChangeTime(300);
   serialConnectionBuilder.enableHandshake();
   serialConnectionBuilder.setBaudRate(2400);
  } else {
   InetAddress inetAddress = InetAddress.getByName(ip);
   TcpConnectionBuilder tcpConnectionBuilder = new TcpConnectionBuilder(
      ↪ inetAddress);
   connectionBuilder = tcpConnectionBuilder;
   tcpConnectionBuilder.setPort(port);
```

```
 tcpConnectionBuilder.useHdlc();
 tcpConnectionBuilder.setRawMessageListener(new RawMessageListener() {
  public void messageCaptured(RawMessageData paramRawMessageData) {
   logger.trace("Received␣raw␣message␣with␣" + "\n\t␣Source␣" + paramRawMessageData.getMessageSource()
     + "\n\t␣Message␣"
     + (paramRawMessageData.getMessage() != null
      ? HexConverter.toHexString(paramRawMessageData.getMessage())
      : null)
     + "\n\t␣APDU␣" + paramRawMessageData.getApdu());
  }
 });
}
connectionBuilder.setClientId(clientAddress);
connectionBuilder.setLogicalDeviceId(logicalDeviceAddress);
connectionBuilder.setPhysicalDeviceAddress(physicalDeviceAddress);
connectionBuilder.setReferencingMethod(ReferencingMethod.SHORT);
// connectionBuilder.setReferencingMethod(ReferencingMethod.LOGICAL);
connectionBuilder.setResponseTimeout(60000);
if (password != null) {
 SecuritySuiteBuilder securityBuilder = SecuritySuite.builder();
 securityBuilder.setAuthenticationMechanism(AuthenticationMechanism.LOW);
 securityBuilder.setPassword(password.getBytes(StandardCharsets.US_ASCII));
 SecuritySuite securitySuite = securityBuilder.build();
 connectionBuilder.setSecuritySuite(securitySuite);
}
// if SnOjbectMapping is not set it is retrieved when first accessing snObjects
if (initializeSnObjects) {
 // ATTENTION: The subsequent mapping is only true for SWiBi L&G E650
 Map<ObisCode, SnObjectInfo> snObjectMapping = new LinkedHashMap<ObisCode, SnObjectInfo>();
 // put 6000 instead of 7392
 SnObjectInfo value = new SnObjectInfo(7392, 3, 0);
 ObisCode key = new ObisCode("1.1.1.8.0.255");
 snObjectMapping.put(key, value);
 value = new SnObjectInfo(2992, 3, 0);
 key = new ObisCode("1.1.1.8.1.255");
 snObjectMapping.put(key, value);
 value = new SnObjectInfo(3160, 3, 0);
 key = new ObisCode("1.1.1.8.2.255");
 snObjectMapping.put(key, value);
 value = new SnObjectInfo(11200, 8, 0);
 key = new ObisCode("0.0.1.0.0.255");
```

```
   snObjectMapping.put(key, value);
   connectionBuilder.setSnObjectMapping(snObjectMapping);
  }
 }
```

### A.2.11 Batch script for Reading Meter Billing Data

```
@ECHO OFF
ECHO ***********************
ECHO Read Meter Billing Data
ECHO ***********************
ECHO -
CALL "%~dp0.\COMPORTS.BAT"
CALL "%~dp0.\IMPORT.BAT"
SET PCALL="%~dp0..\Tools\CallMDU.exe"
SET SCRIPT="%~dp0.\scripts\BillingData.bat"
%PCALL% %SCRIPT% %IMPORT% %COMPORTS%
IF errorlevel 1 GOTO End
cd "%~dp0.\results\"
del␣result*.xml␣>nul␣2>&1
ren␣result*.txt␣result*.xml
:End
PAUSE
```

# Abbreviations

**IoT** Internet of Things

**API** Application Programming Interface

**REST** Representational State Transfer

**XML** eXtensible Markup Language

**CSV** Comma-separated Values

**DLMS** Device Language Message specification

**COSEM** COmpanion Specification for Energy Metering

**ANSI** American National Standards Institute

**OBIS** OBject Identification System

**AP** Application Process

**AL** Application Layer

**ACSE** Application Control Service Element

**ASO** Application Service Object

# Bibliography

[1] Company Profile . http://www.snt.at/about_us/company/company-profile.en.php. Accessed: 2018-03-10.

[2] List of Standard OBIS Codes and COSEM Objects . https://www.dlms.com/documentation/listofstandardobiscodesandmaintenanceproces/index.html. Accessed: 2018-06-24.

[3] OBIS Names, What are They? . https://www.dlms.com/faqanswers/questionsonthedlmscosemspecification/obisnameswhatarethey.php. Accessed: 2018-06-24.

[4] Smart Grids and Meters . https://ec.europa.eu/energy/en/topics/markets-and-consumers/smart-grids-and-meters. Accessed: 2018-06-24.

[5] Support additional protocols for IoT Hub. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-protocol-gateway. Accessed: 2018-02-10.

[6] The DLMS Communication Survival Kit. https://icube.ch/DLMSSurvivalKit/dsk1.html. Accessed: 2018-02-15.

[7] Amazon. Amazon Web Services (AWS). https://aws.amazon.com/. Accessed: 18/05/2018.

[8] American National Standards Institute. American National Standard For Utility Industry End Device Data Tables . Tech. rep., National Electrical Manufacturers Association , 2009.

[9] Companion Specification for Energy Metering . Dlms/cosem architecture and protocols. Tech. rep., DLMS/COSEM, 2007.

[10] Eoghan McKenna, Ian Richardson, M. T. Smart meter data: Balancing consumer privacy concerns with legitimate applications. *Energy Policy 41*, 10 (2012), 807–814.

[11] Federal Energy Regulatory Commission . Assessment of Demand Response and Advanced Metering . Tech. rep., Federal Energy Regulatory Commission, 2008.

[12] Google. Google Cloud. https://cloud.google.com/. Accessed: 18/05/2018.

[13] Gurux. Dlms/cosem meters. https://www.gurux.fi/DLMSCOSEMFAQ. Accessed: 10/04/2018.

[14] Haseeb, S., Hashim, A. H. A., Khalifa, O. O., and Ismail, A. F. Connectivity, interoperability and manageability challenges in internet of things. *AIP Conference Proceedings 1883*, 1 (2017), 020004.

[15] Hoefling, M., Heimgaertner, F., Fuchs, D., and Menth, M. JOSEF: A Java-Based Open-Source Smart Meter Gateway Experimentation Framework. In *Proceedings of the 4th D-A-CH Conference on Energy Informatics - Volume 9424* (New York, NY, USA, 2015), EI 2015, Springer-Verlag New York, Inc., pp. 165–176.

[16] IBM. IBM Watson. https://www.ibm.com/watson/. Accessed: 18/05/2018.

[17] International, S. E. DLMS / COSEM for Smart Metering. https://www.metering.com/top-stories/dlms-cosem-for-smart-metering/. Accessed: 21/05/2018.

[18] International, S. E. Exploring ANSI Standards in Meter Communications. https://www.metering.com/regional-news/north-america/exploring-ansi-standards-in-meter-communications/. Accessed: 22/05/2018.

[19] Janardhana, S., and Shashikala, M. S. D. Challenges of Smart Meter Systems. In *2016 International Conference on Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT)* (Dec 2016), pp. 78–82.

[20] Kursawe, K., and Peters, C. Structural Weaknesses in the Open Smart Grid Protocol. In *2015 10th International Conference on Availability, Reliability and Security* (Aug 2015), pp. 1–10.

[21] Kursawe, K., and Peters, C. Structural weaknesses in the open smart grid protocol. Cryptology ePrint Archive, Report 2015/088, 2015. https://eprint.iacr.org/2015/088.

[22] Mar, W. IoT Hub Service on Microsoft Azure. https://wilsonmar.github.io/iot-hub/, 2017.

[23] Markham, N. *Java Programming Interviews Exposed*. EBL-Schweitzer. Wiley, 2014.

[24] Microsoft. Azure IoT Hub. https://azure.microsoft.com/en-us/services/iot-hub/. Accessed: 01/07/2018.

[25] Microsoft. Microsoft Azure. https://azure.microsoft.com/en-us/. Accessed: 13/05/2018.

[26] openmuc. JDLMS Overview. https://www.openmuc.org/dlms-cosem/, 2013.

[27] OSGP. Open Smart Grid Protocol. http://www.osgp.org/en. Accessed: 18/05/2018.

[28] Saleh, M. S., Althaibani, A., Esa, Y., Mhandi, Y., and Mohamed, A. A. Impact of Clustering Microgrids on their Stability and Resilience during Blackouts. In *2015 International Conference on Smart Grid and Clean Energy Technologies (ICSGCE)* (Oct 2015), pp. 195–200.

[29] Snyder, A. F., and Stuber, M. T. G. The ANSI C12 Protocol Suite - Updated and Now with Network Capabilities. In *2007 Power Systems Conference: Advanced Metering, Protection, Control, Communication, and Distributed Resources* (March 2007), pp. 117–122.

[30] S&T. Supplier of Systems Featuring Proprietary Technologies. http://www.snt.at/index.en.php. Accessed: 10/05/2018.

[31] user association, D. How is DLMS/COSEM Different from Other Standards. https://www.dlms.com/faqanswers/generalquestions/howisdlmscosemdifferentfromotherstandards.php. Accessed: 13/05/2018.

[32] user association, D. What are the Benefits of DLMS/COSEM. http://www.dlms.com/faqanswers/generalquestions/whatarethebenefitsofdlmscosem.php. Accessed: 13/05/2018.

[33] user association, D. What is COSEM? http://www.dlms.com/faqanswers/generalquestions/whatiscosem.php. Accessed: 03/05/2018.

[34] Wang, J., and Leung, V. C. M. A Survey of Technical Requirements and Consumer Application Standards for IP-based Smart Grid AMI Network. In *The International Conference on Information Networking 2011 (ICOIN2011)* (Jan 2011), pp. 114–119.

[35] Watson, I. Watson IoT Platform Gateway Capabilities Now Generally Available. https://developer.ibm.com/iotplatform/2016/04/04/watson-iot-platform-gateway-capabilities-now-generally-available/. Accessed: 02/07/2018.

[36] Weiss, M., Helfenstein, A., Mattern, F., and Staake, T. Leveraging Smart Meter Data to Recognize Home Appliances. In *2012 IEEE International Conference on Pervasive Computing and Communications* (March 2012), pp. 190–197.

[37] Zhang, X. M., and Zhang, N. An Open, Secure and Flexible Platform Based on Internet of Things and Cloud Computing for Ambient Aiding Living and Telemedicine. In *2011 International Conference on Computer and Management (CAMAN)* (May 2011), pp. 1–4.

[38] Zhu, Q., Wang, R., Chen, Q., Liu, Y., and Qin, W. IOT Gateway: BridgingWireless Sensor Networks into Internet of Things. In *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (Dec 2010), pp. 347–352.

# Ramez Elbaroudy

Julius-Raab-Straße 10, 4040 Linz, Austria

ramezemadaiesec@gmail.com  Github  +43 664 9729107

## PERSONAL DETAILS

- **Date of Birth:** January 01, 1995
- **Place of Birth:** Cairo, Egypt
- **Nationality:** Egyptian

## EXPERIENCE

**Super Eddy** **Berlin, Germany**
*Software Engineer Intern* *July 2016 - September 2016*
- Developed a user tracking application for Admin-side tracking, using Node.js and Express.js.

**Unplugged web** **Cairo, Egypt**
*Software Engineer Intern* *July 2015 - August 2015*
- Developed a web application using PHP(Laravel) for the backend of the application.

**German university in Cairo** **Cairo, Egypt**
*Junior Teaching Assistant* *February 2015 - February 2016*
- Taught labs of four courses including introduction to computer science and data structures and algorithms.

## PROJECTS

**GratefulReminder** *April 2018 - May 2018*
- Android app that reminds you with the things you are grateful for in random times.
- Technologies used are RxJava 2, Dagger, Room Persistence Library, Mockito.
- Implemented using MVP design pattern and test-driven development process.

**Codeforces solution explained** *April 2018 - June 2018*
- A collection of Codeforces solutions with explanation implemented in Java and Python.

**Pervasive smart-watch application** *December 2017 – February 2018*
- Android app that collects data using accelerometer sensor of smart watch.
- Using K nearest neighbor to classify whether user is walking, standing or sitting.
- Determine the status of the room by analyzing the context of other users activities.

## EDUCATION

**Johannes Kepler University** **Linz, Austria**
*M.S. in Computer Science* *September 2017 - July 2018*

**German University in Cairo** **Cairo, Egypt**
*B.Sc. in Computer science and Engineering.* *September 2012 - July 2017*
Cumulative Grade: Excellent. GPA: 1.5 (German Scale)

**University of Rostock** **Rostock, Germany**
*B.Sc project and thesis. Thesis Grade: Excellent* *March 2016–June 2016*

## ACTIVITIES

- Participated in ACM Egyptian Collegiate Programming Contest 2014 and 2015.
- Team leader of outgoing volunteer team in, non-profit organization, AIESEC in GUC, Cairo 2017.

## Languages and Technologies

**Expert:** Java, Python, Git, Android; **Prior Experience:** JavaScript, SQL, HTML/CSS, Rails

## Spoken languages

Arabic (Native); English (Fluent); German (very good)

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder inhaltlich entnommenen Stellen deutlich als solche kenntlich gemacht habe.

Linz, Juli 2018 Ramez Elbaroudy