FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Java API and Web Service Gateway for wireless M-Bus

Bakkalaureatsarbeit

zur Erlangung des akademischen Grades

Bakk. techn.

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Ralph Hoch
Matrikelnummer 0405156

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Markus Jung

Wien, 24. Oktober 2013

(Unterschrift Verfasser/in)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Java API and Web Service Gateway for wireless M-Bus

Bachelor Thesis

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Ralph Hoch
Registration Number 0405156

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Markus Jung

Vienna, 24. Oktober 2013 _____ _____

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ralph Hoch
Neulerchenfelder Strasse 87/33, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser/in)

Abstract

M-Bus Systems are used to support the remote data exchange with meter units through a network. They provide an easy extend able and cost effective method to connect many meter devices to one coherent network. Master applications operate as concentrated data collectors and are used to process measured values further. As modern approaches facilitate the possibility of a Smart Grid, M-Bus can be seen as the foundation of this technology. With the current focus on a more effective power grid, Smart Meters and Smart Grids are an important research topic.

This bachelor thesis first gives an overview of the M-Bus standard and then presents a Java library and API to access M-Bus devices remotely in a standardized way through Web Services. Integration into common IT applications requires interoperable interfaces which also facilitate automated machine-to-machine communication. The oBIX (Open Building Information Exchange) standard provides such standardized objects and thus is used to create a Web Service gateway for the Java API.

Contents

1 Introduction **1**

2 M-Bus Standard **3**
2.1 Introduction . . . 3
2.2 History . . . 3
2.3 Purpose . . . 5
2.4 Related Standards . . . 14
2.5 Conclusion . . . 15

3 oBIX Standard **17**
3.1 Introduction . . . 17
3.2 oBIX functions . . . 18
3.3 oBIX examples . . . 19

4 Wireless M-Bus Java Library and API **21**
4.1 Java Library Architecture . . . 22
4.2 Telegram structure . . . 26
4.3 Java Library Usage . . . 28
4.4 Integration into oBIX based IoTSyS framework . . . 32

5 Conclusion **37**

List of Figures **38**

List of Tables **38**

Listings **38**

Bibliography **41**

CHAPTER 1

Introduction

Current power supply systems do not allow a fast and flexible adaptation and are therefore not able to cope with modern approaches for energy conservation. Smart meter devices are the foundation of a newer, smarter energy grid and facilitate the power authorities to deal with on-demand, real-time power consumption values. In Austria, a legal ordinance has been published were general conditions are specified. These conditions include that 70% of all domestic homes have to be equipped with smart meters by the year 2017 and that power suppliers have a monitoring and reporting obligation (Intelligente Messgeräte - Einführungsverordnung (IME-VO) and Intelligente Messgeräte - Anforderungs VO (IMA-VO)) [6–8].

Data exchange between house holds (smart meter) and transaction server components (energy supplier) is realised via power line communication (PLC) using a standard on advanced metering infrastructure [31].

However, PLC only allows a very static data handling approach and it is difficult to apply security arrangements that satisfy modern technology standards. Considering that there are also other issues, like interference with PLC frequency bands, another approach would be desirable.

Additional modules for smart meter devices allow communication via the M-Bus wireless standard [20]. This enables the utilization of TCP/IP technology standards and furthermore allows a near real-time data processing [29]. This is, especially for end consumers, a major advantage, because it provides the opportunity to see effects and results of electronic devices almost immediately. For the energy supplier, a reduction of communication complexity as well as reusing or adapting already existing security mechanism would allow a simpler and more efficient integration with other parts of the energy system. As a bonus it would enable the near real-time tracking of energy consumption data.

To achieve this task, a framework is needed that establishes secure data exchange between smart meter devices and high-end systems utilizing the advanced metering infrastructure.

This bachelor thesis presents a JAVA library and API that enables remote interaction with wireless M-Bus devices. Various functions, such as retrieving and storing values from data

endpoints or support of different telegram types, are supported by the API. In addition, encrypted as well as plain data telegrams are supported. Furthermore, a Restful Web Service has been implemented. It makes use of oBIX (Open Building Information Exchange) [28] to represent the wireless M-Bus meters and serves as a gateway between applications and the meter itself.

The remaining parts of this bachelor thesis are organized in the following manner. Section 2 gives an overview on the current M-Bus standard and especially puts focus on how communication between devices is realized. In addition, a short introduction to the lower level parts of the standard is given. Section 3 shows the oBIX standard and describes how communication can be realized following this approach. In Section 4, the final prototype and its functionality is specified, including examples on how it can be used to interact with meter devices. Additionally, a description of how telegrams are transferred in a specific set up is given. Finally, Section 5 provides a conclusion for this thesis.

CHAPTER 2

M-Bus Standard

2.1 Introduction

M-Bus or Meter-Bus is a standardized system which was developed to support the remote, through a network, read-out of various utility meter systems. These meter systems include heat, electricity and various other units. Remote access allows an M-Bus master to access and retrieve the measured values of the meters. End devices such as laptops or tablets are then used to further process the retrieved data.

2.2 History

In the early 1990s, since there was no system available that enabled remote access of meter device read-outs, a new approach had to be specified. Professor Dr. Horst Ziegler of the University of Paderborn developed an idea for a distributed metering bus system. In cooperation with companies such as Texas Instruments Germany GmbH and Techem GmbH a first prototype was developed. Initially, the idea was to provide a specific physical definition on how the meters can be accessed as well as a protocol specification on how the data should be interpreted.

This effort was later standardized and became a European Standard. The standard can be found under **EN 13757: Communication systems for meters and remote reading of meters** and consists of several parts. The following list gives a short introduction:

- EN 13757-1: Data exchange [14]
  The first part of the standard describes the data exchange and communication of meters as well as the remote read-out of meter data. It provides application specific information and gives an overview of the used communication model.

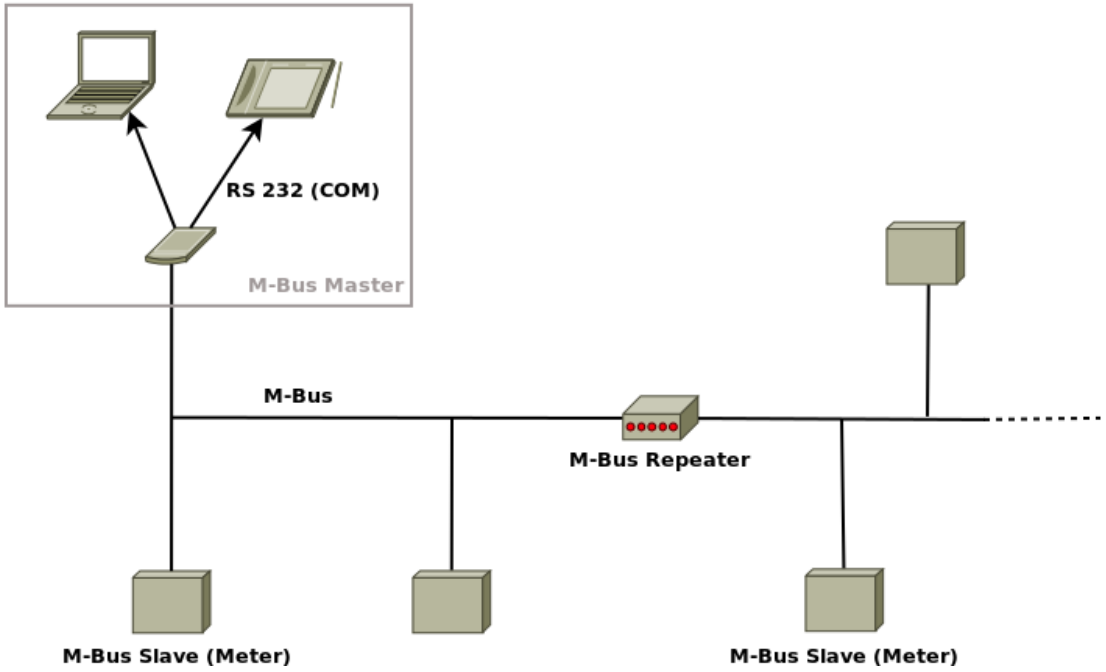

Figure 2.1: M-Bus Standard Setup

- EN 13757-2: Physical and Link Layer [15]
  In this part the physical and link layer are covered. It describes how baseband communication in meter systems is realized over twisted pair media.

- EN 13757-3: Dedicated application layer [16]
  An application protocol on how meter read-outs have to be interpreted is defined. This protocol makes meters from different vendors interoperable. The application layer utilizes the physical and link layer described by EN 13757-2.

- EN 13757-4: Wireless meter readout (Radio meter reading for operation in the 868 MHz to 870 MHz SRD band) [13]
  As modern approaches often use wireless communication, a standard has been defined to support this approach. It defines the physical and link layer for wireless devices and refers to EN 13757-2.

- EN 13757-5: Wireless Relaying [12]
  This part can be seen as an extension to EN 13757-4 and describes how routed wireless networks for meter read-outs can be established. As meter and bus master devices can be physically located far apart from each other, a relaying can be necessary.

- EN 13757-6: Local Bus [11]
  This standard describes parameters of the electronic link layer of a local bus system for the communication with meters within the bus system.

The M-Bus standard is closely related and based on the ISO-OSI reference model. An advantage to base this new system on an already existing, standardized open system is that it allows an incorporation with various already established network protocols. For further details, please refer to Section 2.3.

2.3 Purpose

The key goal of meter bus systems is to provide an easy extendable, stable and cost effective way to interconnect many devices (slaves) over a long distance with a master application like a data center. [31]

From the existing network topologies (star, ring, bus) only a bus system is suitable for this task as it allows serial transmission of data over a common used transmission medium as well as easy addable new devices. Meter systems are used to measure the consumption of various resources like heat or electricity. Furthermore, they deal with end user data which is often used to provide billing and therefore the receiving of accurate data is indispensable. Thus, transmission integrity is very important and the bus system needs to be insusceptible against external interference such as inductive interference. [21]

As there are probably many end devices used in such a bus system it is imperative to keep the operating costs as well as the installation costs as small as possible. In calculating the costs, it is necessary to keep in mind that the transmission distance may be very long and thus the transmission medium needs to be cost effective (e.g. standard medium, no additional shielding). Complexity of installation of new end devices should be kept simple, as it might be necessary to add devices or facilities during bus system operation. Remote powering enables meter slaves to be independent.

Inaccuracies in the existing standard enabled manufactures to develop slightly different meter device specifications and as a result incompatible units were produced. Installation of new devices had to be approved and tested for compatibility with the existing system and devices.

During the smart-metering initiative of the European Union these inaccuracies led to the organization of an Open Metering System. An Open Metering Specification (OMS) was defined where the standard interpretation from different manufactures had been specified to make them interoperable. Additionally, a 128-Bit AES encryption was introduced for the powerline communication as well as the wireless transmission. As of 2009, OMS has been proposed to the European Committee for Standardization as a supplement to the existing standard.

The ability of the M-Bus system to deal with many meter devices makes it the foundation of modern energy grids. The bi-directional communication method enables remote control of the meter slaves that are in place. The near-field communication of modern meters is often realized utilizing the M-Bus system.

In combination with Smart Meters a near real-time transmission of power consumption data to power supply companies can be established. The term “Smart Meter” usually refers to electricity meters, but can be used for other energy sources like water or gas as well. Using Smart Meter devices has multiple benefits for both the consumer and the power supply industry (power supplier, grid company, service providers).

On consumer side, it gives more transparency on actual costs and it can provide near real-time consumption data for single household appliances like TVs or computers. This up-to-date measurement data supports the consumer in a more efficient usage and control of his energy use. Furthermore, it can be used to identify high energy consumption devices and help to reduce the overall energy wastage. Another advantage is that the consumer doesn't have to rely on statistically calculated billing information. The always up-to-date power usage data allows an exact billing information on the actual energy consumption and thus gives the consumer more transparency on costs.

The power supply industry shares some benefits with the consumer as they can act on the effective power consumption of each consumer individually. Monitoring energy consumption provides them with a better understanding of the actual total power consumption in their region. Considering systems that monitor and compare data with statistically evaluated data, it is possible to develop a remote fraud detection system and help to increase the safety of consumers. Especially useful for power suppliers is the combination of smart meters with an advanced metering infrastructure (AMI) as it allows two-way communication [30].

This type of communication supports the remote activation or deactivation of devices as meters can receive commands through a network. Remote access allows a variable, adaptable through-put of meters to react to specific energy demands. The remote activation of meters also makes it more cost effective as no physical installer is necessary and electricity contracts can easily be switched between customers.

Custom pay rates, similar to telecommunication networks, and dynamic pricing based on time or supply constraints and introduction of new value added services are possible.

Utilizing the advanced metering infrastructure, an accumulation of energy grid participants, including power suppliers, grid companies, service providers and consumers, can build a Smart Grid. Through the behaviour and the resulting information of all involved parties an intelligent energy grid can be developed that is able to react according to energy demands or availability. In the process, a more stable, reliable, effective and efficient energy network is formed. [18]

Protocol Overview

The M-Bus protocol stack is based on the OSI-Model (Open Systems Interconnection) of the International Organization of Standardization (ISO). This model standardizes a communication system by using several abstraction layers where each layer adds information to the previous layer. The OSI-Model consists out of seven layers that form a protocol stack where each layer utilizes the layer underneath it. Since M-Bus does not provide all features of an ISO-OSI computer network, only a subset of this protocol stack needs to be used [4]. The M-Bus protocol

stack, thus, only uses three layers and builds upon the standard defined by IEC EN 61334-4-1. Table 2.1 gives an overview of the stack.

Number	Layer	Function	Standard
7	Application	5	EN 13757-3
2	Data Link	8	EN 13757-2 or EN 13757-4
1	Physical	4	EN 13757-2 or EN 13757-4

Table 2.1: M-Bus Protocol Stack

An advantage of utilizing this protocol stack is that higher protocol layers are independent from their low level counterparts as communication is only realized between layers of the same level. This means that the implementation of the lower level layers can be changed and the functionality of the system is not compromised. Applying this configuration allows for example the usage of wired (EN 13757-2) as well as wireless (EN 13757-4) communication media without any changes to the actual application. Defining a protocol for these layers allows interchangeability of meters. Special gateways/bridges are in place to connect network parts with different protocol implementations.

Communication

Communication is always established between two types of devices. In case of M-Bus this is the master and the slave (meter device). Usually, the master (caller) calls the slave (called) to retrieve the measured data and during communication each side keeps their function, hence is caller or called. Using this kind of denotation is analogue to the common client/server architecture that is commonly used in network systems. In most cases the caller can be seen as the client that requests a functionality from the server (called). However, it does not necessarily need to be this way. The bi-directional communication enables the meter device (server) to call the master bus (client). In case of a malfunction, an error or an alert the meter can therefore inform the master on this unexpected behaviour.

As communication can last over a longer period of time the partition into transactions is reasonable. Each transaction can be viewed as a request from the caller to the called. During transactions caller and called alternately receive and transmit data. The following example will demonstrate this behaviour schematically:

- Client (caller) sends (transmits) a request command to the server (called)
- Server listens for incoming commands and receives them as soon as something is transmitted
- Server responds (transmits) the result of the executed command
- Client listens for response (receive)

Physical Layer

The physical layer specifies the lowest protocol level and deals with how data is actually transmitted over the transfer medium. It is possible to have different kind of transports at this level and thus it is necessary to define how bits are represented and interpreted by the communication participants. The preferred method that is proposed by the EN 13757-2 standard [15] is baseband communication over twisted pair (M-Bus). Figure 2.2 shows the physical layer definition of [15].

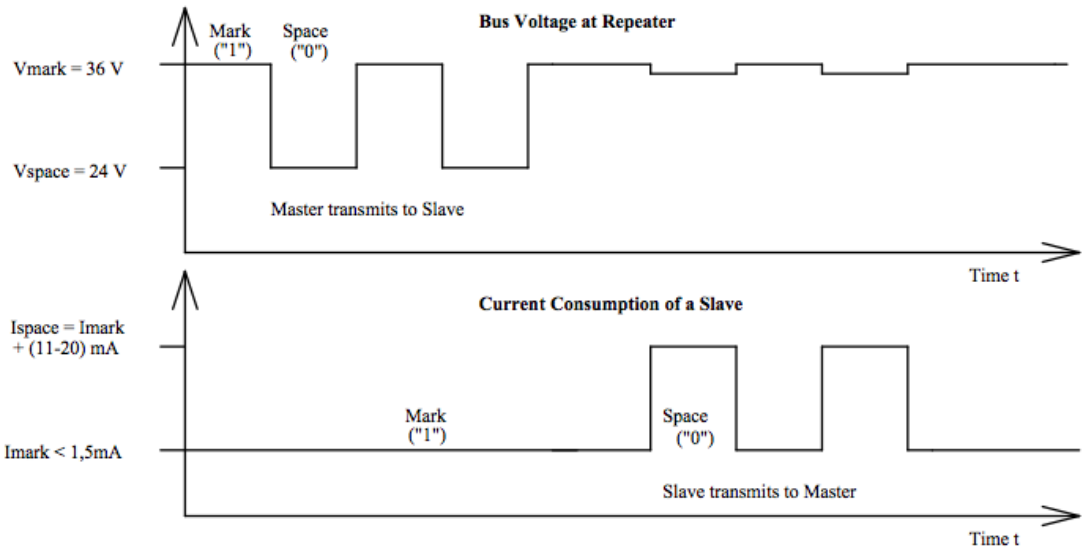

Figure 2.2: Physical Layer - Bit representation

As shown in Figure 2.2, transfer of bits is accomplished by voltage level changes on the master side. Transferring a logical 1 from the master to the slave is realized by using a nominal voltage level of +36 volts, a logical 0 is represented by +24 volts. Slaves or meters are powered remotely and take the required current from the bus system. The defined protocol allows a two-way communication, however, at the same time transmission is only possible in one direction. If the slave initiates a transfer it is accomplished by modulating the consumption (current) of the slave device. Current up to 1.5 mA represent a logical “1” (Mark) in this case and a logical “0” (Space) is represented by an additional current consumption of 11-20 mA. For a full in depth description of the bit representation and transportation in the physical layer, please refer to [9] and [15, p. 8].

The implementation of the physical layer can change and therefore it is possible to exchange the underlying transfer medium with another medium. As the data link layer is closely related to the physical layer, both of them are described in the same standard.

As a matter of principle, it is possible to use a different implementation of the layers defined in [15]. For further details please refer to alternatives discussed in EN 13757-1 [14].

Data Link Layer

Communication is realized by using several transactions. As mentioned before transfer is only possible into one direction at a time, thus a method to synchronize communication is essential. Furthermore, as an M-Bus system can contain several slaves, it is necessary to somehow identify (address) the participating meters.

This layer specifies a controlled connection between a master and a slave and is closely related to the physical layer. Transmission of data is carried out by serial, asynchronous bit transfer. Synchronization can be achieved by defining how characters are transferred. Characters are transmitted one at a time and always consist out of eleven bits. Each character begins with a start bit (space) and is followed by eight data bits, a parity bit (check for equality) and a stop bit (mark). The start and stop bits mark the beginning and the end of a character and support synchronization. [15]

To send meaningful data, characters are organized in defined telegrams which are shown in Table 2.2.

Single Character	**Short Frame**	**Control Frame**	**Long Frame**
E5h	Start 10h	Start 68h	Start 68h
	C Field	L Field = 3	L Field
	A Field	L Field = 3	L Field
	Check Sum	Start 68h	Start 68 h
	Stop 16h	C Field	C Field
		A Field	A Field
		CI Field	CI Field
		Check Sum	User Data (0-252 Byte)
		Stop 16h	Check Sum
			Stop 16h

Table 2.2: Data Link Layer - telegram formats M-Bus

Each row of Table 2.2 represents one byte (plus the bits for start, parity and stop). Some of these bytes can have special meanings and are used to control transfer or address participants:

- C (Control) Field
  supports determination of communication direction. Additionally, it encodes the function

of the telegram. For a complete list of functions please refer to [4] as this field is quite comprehensive. An example for a control value is:
SNDNKE: 40h short frame - initializes slave

- A Field (primary address)
  has the purpose to identify (address) the communication endpoint and is used by both the sender and the receiver. As this field is only one byte long it is theoretical possible to address 256 communication participants at max. However some addresses have a special purpose and can't be used for devices [16, p. 67]. This limits the directly addressable devices to 250.
- CI Field (Control Information Field)
  is from high importance for the application layer and is used to distinguish between long and short frame telegrams.
- L (Length) Field
  defines number of bytes of the meter data (including C, A and CI Fields).
- Check Sum
  checks if the transmission has been correct. It is configured by specific parts of each telegram and serves as an additional protection for synchronization or transmission failures. The standard defines CRC (cyclic redundancy check) as the used method.

In addition, a short explanation of the different telegram formats is given:

- Single Character
  a single chracter is used to acknowledge that the telegram has been received (ACK: E5h).
- Short Frame
  a short frame has a fixed size of five bytes and has a specific format. The function is controlled via the C field and participant is resolved by the A field. For example, it is used by the master to initialize transmission of measured values from a participating meter.
- Control Frame
  conforms to long frames. However, no user data is transferred.
- Long Frame
  compared to the short frame a long frame can also contain user data of variable length. Control and address fields are used synonymously as in the short frame and also a check sum is calculated based on the contained fields.

Application Layer

The application layer serves as the final tier for the user. Custom applications need to understand the protocol at this level to be able to extract meaningful data from consumer utility meters.

Usually applications work similar to client/server applications where the master device, this could for example be a computer, tablet or laptop, acts as a client and the utility meter acts as a server. This means that applications send requests to the server and then await the response. Although it is possible to have communication initialized by the utility meter, the other way around is the standard communication pattern.

As mentioned in Section 2.3 telegrams consist out of several fields - depending on the telegram. For this layer, the CI-Field is particular interesting as it encodes the type and sequence of the data that is transferred in the telegram. There are two modes in operation and depending on how the mode bit is set, the data gets interpreted (if the mode bit is set the most significant byte gets transferred first, otherwise the least significant bit is transferred first).

Additionally the CI-Field defines what is currently transmitted. As all telegrams in the application (and higher level) layers are based upon variable user data the length has to be specified in the frame of the transmission (data link) layer. Data is transferred as a telegram inside the frame from the lower protocol level and consists out of a telegram header and the actual data. The telegram header can be set in different formats depending on the purpose of the transmission [16]. Depending on the header several control values can be set and used to address different kinds of operations. For example, it is possible to specify secondary addresses to identify the slave entity (in this case the meter device) or mark a synchronization package. As only 250 devices are directly accessible, the secondary address allows to expand the address space. In case a secondary address is used, the device with this "unkown" primary address gets bound to the primary address 250 which is specifically for this purpose. This allows basically to address an arbitrary amount of devices [16, p. 47]. Furthermore, single fields can set the encryption method (signature field) of the telegram or report an alarm notification. Part of the header is also an identification, manufacturer and other numbers for additional information. For a complete list of possible operations see the official specification [16]. Another important task of the header is to denote if the telegram is a request or response.

Lately, there has been a lot of effort to move M-Bus systems towards a wireless based communication topology. As the current official norm only has rudimentary support of wireless communication systems a new draft of the norm had to be formulated. As of 2012, this draft is currently under review and waiting for approval. Several changes in the specification layer have been introduced and some of them are targeting specifically the Wireless-M-Bus systems. A specific header configuration is necessary for wireless communication and this header must contain at least:

- status byte
- configuration word
- access number

Other changes include (among others):

- additional encryption formats
- telegram headers for wireless communication systems

- more data points for electrical measurement units

Applications for end users (consumers) should build upon this level and provide a simple interface to query data and display it accordingly. This layer is the final layer of the protocol stack but it has to be kept in mind that only the protocol is defined and applications for end consumers have to implement this protocol.

As the application layer protocol is quite complex there is not enough space to discuss it here. For a complete evaluation please refer to [16] and [4].

Wireless M-Bus

Using the OSI-Model as an reference model has the advantage that implementations of layers can be changed and the overall system still is operational. This allows new technologies to be put into place to replace existing ones. Furthermore, it is possible to interconnect different kinds of implementations by providing a bridge for each network part so that communication can be established throughout the network.

Modern approaches often rely on wireless connection of devices as it is very cost effective. The same applies also for M-Bus systems and thus a specification for a Wireless-M-Bus system has been formed and its result has been accepted as an European norm. The norm specifies how the physical and data link layer for wireless communications operate and is specifically targeting short range devices in unused frequency bands [13].

Wireless-M-Bus systems allow communication between measurement entities and non stationary units (for example, master devices such as a laptop as a data collector). To achieve the communication, several operation types are specified (all operation types are identified by a name and a number):

- stationary operation method (S):
  This method is used for the unidirectional or bidirectional data transmission between measurement units and flexible master devices. Sub-methods of this type include optimized methods for long message headers (S1) and mobile devices (S1-m).

- frequent send operation method (T):
  Here small telegrams are used to transfer data in very short time frames (seconds). This allows to track measurement data in a very short time and is utilized by mobile devices that are not constantly in range of the meter itself. Furthermore, this method allows to create a measurement graph on almost real-time data.
  Sub-methods include operations for only sending data in periodic time frames or at random (T1) and a bidirectional method that uses a short initialization telegram to create a transmission channel (T2).

- time frequent receive operation method (R):
  A measurement unit listens for incoming messages (in a frequent interval). If it receives one it issues a transmission channel with the sender. This method allows the master device to query several meter units at the same time as all of them use a separate frequency channel.

It is possible to combine these operation types and therefore use more than one at once. The devices must be built to support all types if they are used in different systems.

Often these operation types share the same configuration of the physical and data link layer to allow interoperable applications. However, specific configuration for these layers as for example encoding methods can occur and are specified in the European norm [13].

Different types of operation have different constraints. This means that for all methods other operations of establishing a connection apply and also that they have varying time frames during transmission. Meter devices usually have a specific delay after a response until they are able to process the next incoming request.

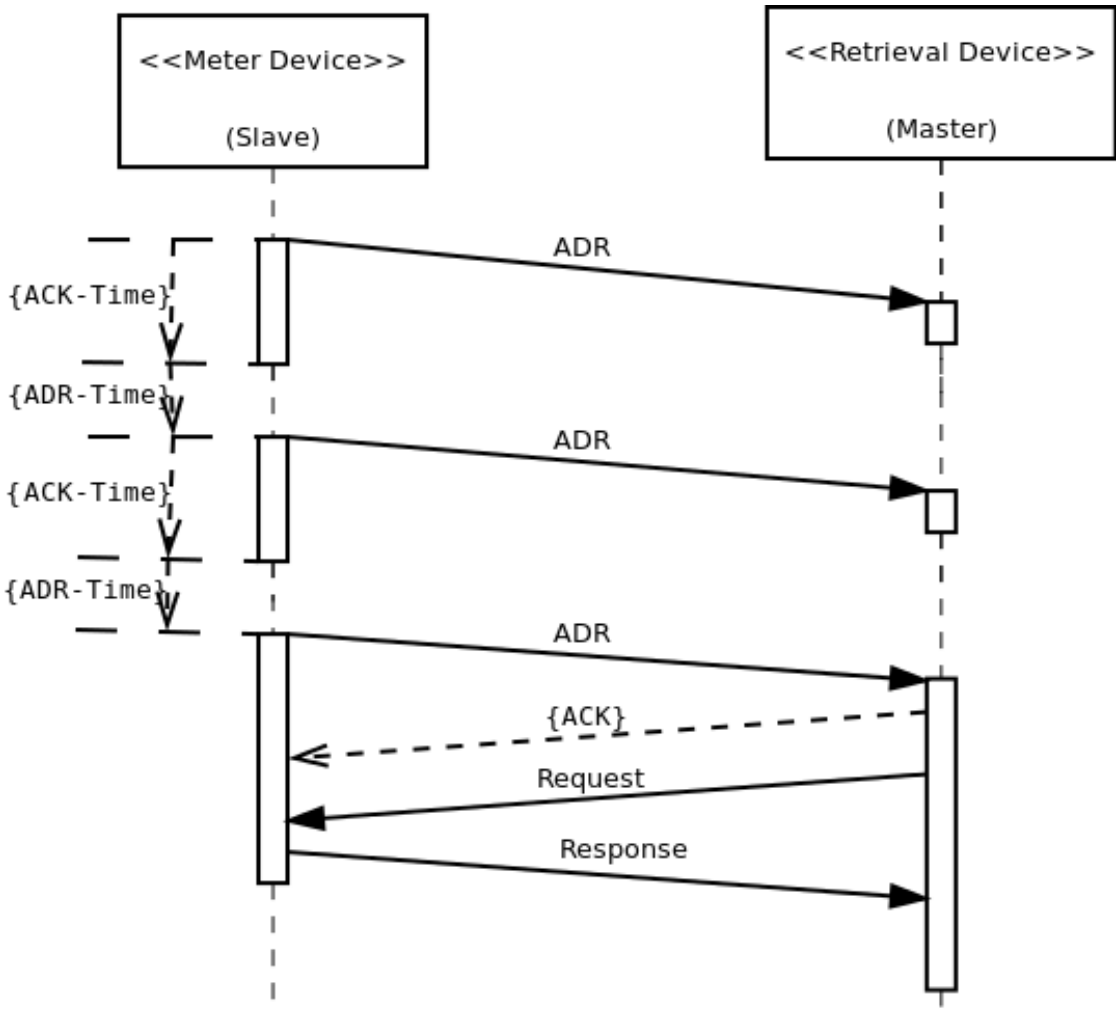

Figure 2.3: Wireless-M-Bus communication - Simplified bidirectional communication (as in T2)

Figure 2.3 shows an example for a data exchange between a master and a slave unit. The communication mode used is a bidirectional method that is based on a short initialization frame (ADR - Access Demand Request) that is sent in periodical intervals (T2). This frame is sent out by the meter (slave) to be detectable for retrieval devices (master). After the slave device sends such an ADR it waits for a specific time frame to receive an acknowledgement packet (ACK-Time). If such an acknowledgement is received a bidirectional communication channel is opened, otherwise the slave issues a time-out until the next ADR is sent. In case an acknowledgement packet is received at a master device the slave is ready to receive requests from the user side (master). After receiving a request the slave answers with the corresponding response and continues with its normal operation. This example simplifies the process as there are several additional constraints that apply for the communication like considering response time or time-outs for not receiving requests after an acknowledgement [32] [13].

The current standard only covers the basic operation methods described above and is not sufficient for modern purposes any more. Thus, a new standard has been described and is currently under review and will replace the existing European norm.

The new norm introduces several additional operation types that allow new implementation fields (meter devices which operate mainly as receivers of commands for example). Furthermore, a more detailed definition of how the physical layer operates is given. Additionally, the data link layer, and especially how the frames including their control headers are built, is described in more detail.

2.4 Related Standards

The protocol specification described in DIN EN 13757 is not the only approach to standardize interaction between meter and master devices. Another approach is laid down in IEC 62056 where several standards for *Electricity metering – Data exchange for meter reading, tariff and load control* are combined. The IEC 62056 standard is the international standard version of the open DLMS/COSEM (Device Language Message Specification/Companion Specification for Energy Metering) specification and is issued by IEC (International Electrotechnical Commission). It is maintained by the DLMS User Association and is split into several specifications which are published as books. The COSEM provides specifications for the Transport and Application Layer based upon the DLMS protocol [5]. This standard is also available as an European Standard under DIN EN 62056-21 [22].

SML (Smart Message Language) provides a communication protocol for meter devices and allows data exchange in between them and retrieval devices [33]. Several meter devices support this communication protocol such as SyM^2 (synchronous actual usage meter), eHZ (electrical household meter) where SyM^2 is based on an Ethernet connection and eHZ uses an infrared interface according to the standard described above, DIN EN 62056-21. Communication is realized based on small messages which are called files. All messages consist of a start and end sequence. The messages themselves have data stored (e.g. data values or commands). Messages can be sent in a standard text format or in a compressed binary format [25] [24].

A comparison between DLMS/COSEM and SML can be found in [17].

In the North American market often a standard published by the American National Standards Institute (ANSI) is used. ANSI C12.18 describes a communication protocol that is used in a two-way communication with meter devices. This particular standard describes the lower protocol levels and how communication via an ANSI Type 2 Optical Port can be realized. Subsequently, other standards or extensions have been published which extend the original standard by other technologies like for example modems (ANSI C12.21) which are better suited for automatic meter reading. The ANSI C12.22 standard describes a protocol for transporting data specified in ANSI C12.19 (table data) over networks. The purpose is to enable interoperability in between master and meter devices. ANSI C12.22 also supports data transport via TCP or UDP. For detailed information please refer to [2, 3].

2.5 Conclusion

Modern approaches that enable a more economic use of resources are a vibrant alternative to currently established inflexible systems. Smart Grids make a dynamic use of resources possible as they allow to use them on an "as needed" basis. This helps to cut costs which not only the electrical network supplier but also the customers benefit from. A more transparent view on the actual usage of energy provides the customers with the possibility to have more control over their energy consumption and to reduce or eliminate unnecessary units from their daily use.

This is a big advantage and makes home automation possible. Single devices can be addressed individually and even controlled by the consumers themselves. To enable such an automation a technology is necessary that creates the opportunity to communicate with devices through a network. Manual steps, such as reading meter values, are not necessary any more.

M-Bus systems provide such an infrastructure and are an integral part of home automation and thus also a modern Smart Grid. Smart Meter devices are very flexible as they are able to receive commands over a network and operate accordingly. Not only is it possible to receive near real-time energy measurement data but also to control single devices. Applications can be built for various purposes and in doing so allow consumers access to devices that they can control through a simplified user interface.

The idea of such an easy extendible, flexible and cost effective infrastructure has been formed, refined and reached the status of an European norm which now enables interoperability between different vendors. Wireless communication methods are included as well for seminal application scenarios with mobile devices. Refined versions of the EN 13757 norm are already under review and will expand the possible scenarios even more.

However, since this is a relatively new infrastructure, several points remain that need to be clarified. Security mechanisms have to be put into place to help to prevent fraud and ensure the consumer security. Newer versions of the EN 13757 norm already include modern encryption methods that help to secure the data that is transmitted through the infrastructure. How larger-scale applications deal with other security issues related to Smart Grid technologies remains to be seen.

Another point is that currently the infrastructure, from a hardware perspective at the consumer side, is not available. The next step is to bring this infrastructure to the consumer. This will be an ongoing process over a longer period of time.

In conclusion, it may be said that the M-Bus infrastructure can be seen as the foundation of home metering environment and later on of a modern Smart Grid. Smart Meter devices facilitate the possibility of a more economic energy consumption in a modern Smart Grid environment.

CHAPTER 3

oBIX Standard

3.1 Introduction

With the development of modern Internet technologies, the term ubiquitous computing is becoming more and more common. Consumers want to be able to access and control various devices, for example, a home heating system, through gadgets like mobile phones. This means that devices or systems must be remotely accessible through a common infrastructure that support controlling these devices. All these requirements are often subsumed under the term Internet of Things [10, 19].

Only a small amount of systems provide built in remote access and if they do they often rely on custom and proprietary communication infrastructures. Furthermore, low level communication protocols, for example, the M-Bus standard, are very common and prevent an easy to use data exchange between various devices. However, these low level communication protocols, such as the M-Bus standard, provide the foundation for and are used to build applications upon them. Systems such as smart grids are an example for this [34].

To facilitate an easy to use communication between devices a common communication infrastructure is necessary. Web Services are widely accepted and can be used to accomplish this task. In addition, devices that communicate with each other need to "speak" the same language, which means that they have to communicate using a standardized protocol. The oBIX (Open Building Information Exchange) standard tries to fill in this gap with a proposal for a standardized communication protocol for devices. The standard is implemented as a RESTful Web Service infrastructure and uses technologies like XML, HTTP and URIs [28].

Devices that participate in oBIX-based communication can be called oBIX entities and can exchange information with each other through Web Service calls. This means that also automatic machine-to-machine communication can be accomplished which helps the task of home automation in general and smart grids in particular [23]. Since communication is realized through Web Service calls, interaction can be realized with various devices. As an example we can imagine a Web Service that allows the administration of a home heating system throughout modern

smartphones. New devices can be made public by registration in an oBIX based system and then provide additional functionality [27].

3.2 oBIX functions

The main focus of oBIX is to provide a way to communicate with devices. oBIX supports three different types of interaction through a REST based infrastructure: read an object, write an object and invoke an operation. The payload transferred is an XML representation of the oBIX standard. To identify entities or objects URIs (Uniform Resource Identifier) are used that can also be used to mark the location of a device which means that they can also be seen as URLs (Uniform Resource Locator). For an example, please see Section 3.3.

Basically, the oBIX specification provides a small fixed set of object types which can be mapped to XML structures. An oBIX object of type *obj* (which serves as the root element of all oBIX objects), for example, defines several properties like *name* or *href*, to identify the object, among others. All other types are derived from this basic type.

Another key concept of oBIX are the *Contracts*. *Contracts* can be seen as a template object that other objects reference to through an *is* attribute. Properties that reference to such a *contract* can be accessed according to the *contract* specification and thus provide methods accordingly. Furthermore *contracts* can be used to define new types and also allow abstraction through inheritance. This allows you to introduce new types which enrich base types with additional functionality. An object can reference more than one template and templates can have multiple inheritance. This mechanism makes oBIX easy to use as well as easy extendible.

As already mentioned objects can reference each other through URIs. Beside that it is also possible to have nested objects. As an example, we will refer to a home heating system again where the heating system could be presented by an oBIX object which has several heaters as containment properties. These heaters are themselves oBIX objects. In this case, also all objects would follow a *contract* specification.

In home automation usually certain aspects of the systems are of interest. In our home heating example we could have several heaters and of each of them their current temperature setting could be from interest. These values of sensors or similar devices are often referred to as endpoints. In case of oBIX they are identified as *points* and are represented as a *contract* named *obix:Point*. Each *point* has a specific type (bool, real...), a value and a unit. An example can be seen in Section 3.3.

Besides the features mentioned above oBIX provides more complex tasks as well. For example, a concept named *watches* is used to provide real-time information on objects. It is possible to register a service at a *watchService* and receive data updates in near real-time. Another example is the *history* concept which supports querying objects for values from a specific time frame. In case a query is made an XML presentation of a query is send to the service containing the timeslot. The service then answers accordingly (please see 3.3).

Since the main focus of this thesis is not on the oBIX standard only a short overview is provided. oBIX provides more features cannot be covered here. For further details please refer to [28]

3.3 oBIX examples

To give a better understanding of the oBIX communication protocol a few short examples are given here. These examples are chosen to show certain aspects of the oBIX standard and do not intend to cover the whole communication process.

```
<obj href="/smartMeter" is="iot:SmartMeter">
  <real name="energy" href="energy" val="0.458041666666664" unit="obix:units/↩
      kilowatthours"/>
</obj>
```

Listing 3.1: oBIX object SmartMeter which holds an endpoint

Listing 3.1 shows an example of a simple oBIX object. All oBIX objects can be accessed by a unique URI which means that a call of `http://localhost:8080/smartMeter` (assuming that the Web Service is running on localhost with port 8080) would give the result shown in Listing 3.1. In this example it is shown how a certain object, in this case a smart meter object, can be accessed. The object references a contract *iot:SmartMeter* through the *is* attribute and holds itself a value with the name *energy* which is represented as a floating point with value 0.46 and the unit *obix:units/kilowatthours*.

```
<obj href="/smartMeter" is="iot:SmartMeter obix:History">
  <bool name="switchOnOffValue" href="switchOnOffValue" val="false" writable="true"/>
  <real name="power" href="power" val="0.0" unit="obix:units/watt"/>
  <real name="energy" href="energy" val="0.458041666666664" unit="obix:units/↩
      kilowatthours"/>
  <ref name="power history" href="power/history" is="obix:History"/>
  <ref name="energy history" href="energy/history" is="obix:History"/>
  <ref name="power groupComm" href="power/groupComm" is="iot:GroupComm"/>
  <ref name="energy groupComm" href="energy/groupComm" is="iot:GroupComm"/>
</obj>
```

Listing 3.2: complex oBIX object

In an oBIX object, several entities can be registered and accessed through their respective URIs. In this case the URIs also serve as a relative URL and thus allow easy navigation through a common web browser. All entities are accessible through a standardized oBIX interface and provide operations according to their contracts. The smart meter device references the contract *iot:SmartMeter* and can be accessed through the */smartMeter* URI. Listing 3.2 shows an example for such a call. The object is a little bit complexer than the one described in Listing 3.1 and provides several endpoints which can be accessed. In addition, it shows some references to other/included objects. Some of the endpoints in this example are writeable. Writeable endpoints specify that values can be set through a Web Service call (PUT request). Listing 3.3 shows an example on how we can interact with this object.

```
<bool name="switchOnOffValue" href="/smartMeter/switchOnOffValue" val="true" writable↩
    ="true"/>
```

Listing 3.3: oBIX writeable object

After sending the code snippet shown in Listing 3.3 via a PUT request the value of *switchOnOffValue* is set to true.

```
<obj name="history" href="/smartMeter/energy/history">
  <int name="count" href="count" val="100"/>
  <abstime name="start" href="start" val="2013-05-22T21:43:30.298+02:00" tz="Europe/↩
      Vienna"/>
  <abstime name="end" href="end" val="2013-05-22T21:48:27.355+02:00" tz="Europe/↩
      Vienna"/>
  <op name="query" in="obix:HistoryFilter" out="obix:HistoryQueryOut"/>
  <op name="rollup" in="obix:HistoryRollupIn" out="obix:HistoryRollupOut"/>
</obj>
```

Listing 3.4: oBIX history object

In Listing 3.2, it is shown that an object in oBIX can also hold a reference to a *history* object. An example for an oBIX history object can be seen in Listing 3.4. The object specifies how many entries are currently stored in the history (count attribute, in this case 100) and from which timeslot (attribute start and end). In addition, this example also shows how an oBIX object can provide an operation through which specific data can be requested (operation query).

oBIX objects can be quite complex and also provide mechanism for nested objects. This thesis only aims at giving an overview, for an exhausting description of the oBIX specification please see [28].

CHAPTER 4

Wireless M-Bus Java Library and API

The main goal of the implemented prototype was to provide a higher level of abstraction and in doing so providing an API (Application Programming Interface) that is easy to use and allows interacting with a meter device. Thus, the API should provide methods to receive and write telegrams as well as automatically process them. The API allows applications to interact with wireless M-Bus meter devices and process their telegrams in an automatic way. Data extraction functions have been put into place to provide telegram parts in a more meaningful way. In addition, support for encryption and decryption of the communication via the AES (Advanced Encryption Standard) standard has been added. Telegram headers can be parsed and their payload can be processed accordingly. Using this functionality it is also possible to process telegrams with various header formats. Since the idea was only to interact with meter devices only a subset of the entire M-Bus standard has been implemented.

Figure 4.1 gives a schematic overview of the API. It shows the main packages or components of the API and how they are related to each other. The main components are:

- Manager-Component
  This component is used to set up and initialize the overall API. Its basic functionality is to initialize the connector and the telegram manager itself. Although all other components can be used standalone, this component provides an easy way to initialize the API.

- Connector-Component
  To establish communication with a meter device a connector is necessary. This component establishes a connection via a COM-Port and listens in a periodic interval for incoming messages. The messages are then handed to the telegram manager and processed from there.

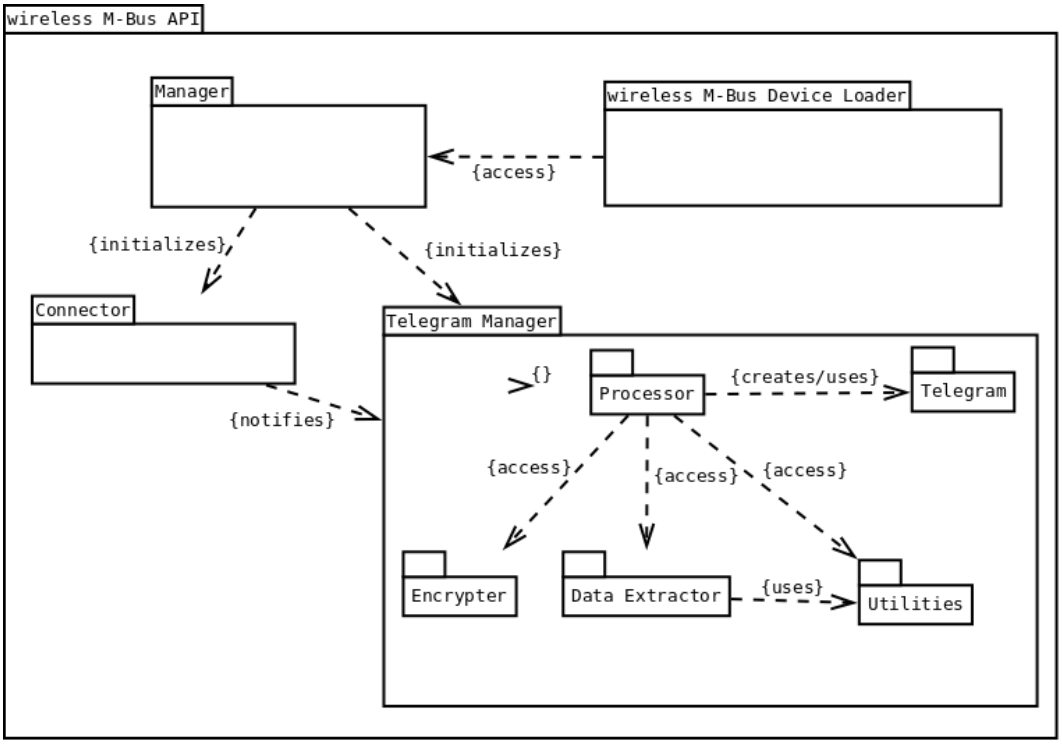

Figure 4.1: Package diagram that provides a schematic overview of the API

- Telegram Manager-Component
  Retrieves incoming telegrams from the connector and initializes the parsing process. Telegrams are stored in raw format as well as in parsed format. In addition, the telegram manager provides several API functions that make the telegram values easy accessible.

- Wireless M-Bus Device Loader-Component
  As the API also provides integration into the IoTSyS framework (Internet of Things integration middleware) it has to provide necessary connecting interfaces. This loader component is used to map meter telegrams according to the oBIX standard so that they can be used in the IoTSyS framework [26].

This schematic overview will be discussed in more detail throughout Section 4.1.

4.1 Java Library Architecture

The prototype has been implemented in the JAVA programming language and provides a test suite as well as a configurable set up for meter devices. To access the virtual COM-Port a third party library RXTX has been used which takes care of handling communication over COM-Ports. To use RXTX, a library has to be registered in the JAVA_HOME/lib folder as well as the

corresponding jar-File needs to be on the classpath. For further installation instructions, please refer to [1].

```
Packages:
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.reader
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams.body
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams.header
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams.util
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.test
  at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.util
```

Listing 4.1: java prototype package structure

Listing 4.1 shows the overall package structure of the prototype and each package may contain multiple classes. The package *reader* contains all classes necessary to receive and send data to a smart meter device. In particular, this means that it listens on a configurable COM-Port for incoming messages. It is realized as a thread based implementation so that messages can be received asynchronously from the overall application. Through a specific listener interface the application is then notified upon the arrival of new information/telegrams. The *telegrams* package consists out of several POJOs (Plain Old Java Objects) among other classes. These classes represent the telegram structure as JAVA objects and provide meaningful methods for accessing distinct parts of the telegram itself. Several of these objects are nested as a telegram consists out of both a header and a body where the body itself consists out of several objects. Since a lot of the fields in the classes correspond to specific parts in the telegram a *util* package has been introduced which takes care of conversion or calculation operations. Additionally, it is used to construct an output that can be easily processed by consumers. Last but not least a *test* package is part of the prototype. Through this package several test cases can be tested without relying on the actual connection to the meter device itself. These tests include conversion, calculation, output and telegram construction. Beside that there are also tests available with an actual meter device.

To give a more detailed view into the internal structure of the API several class diagrams are included in this thesis. Since the overall structure is quite complex the class diagram has been split into several diagrams where each diagram focuses on a particular part.

Figure 4.2 gives an overview of how telegrams are constructed in the API. Telegrams itself are a quite complex construct since they can vary in length and field definitions depending on the header and several other field (more on this can be found in Section 4.2). Telegrams consists out of several distinct parts but in any case they have a *header* of a specific length (which can variable) and a *body*. The *body* usually consists out of another *header* and the *payload* which contains the actual data values. Each field is represented through a class *TelegramField*. As there are multiple fields with different functions the *TelegramField* class can be seen as a template class for several other, more specific, classes. The *payload* of the body usually is made up out of variable data records presented as *TelegramVariableDataRecord*.

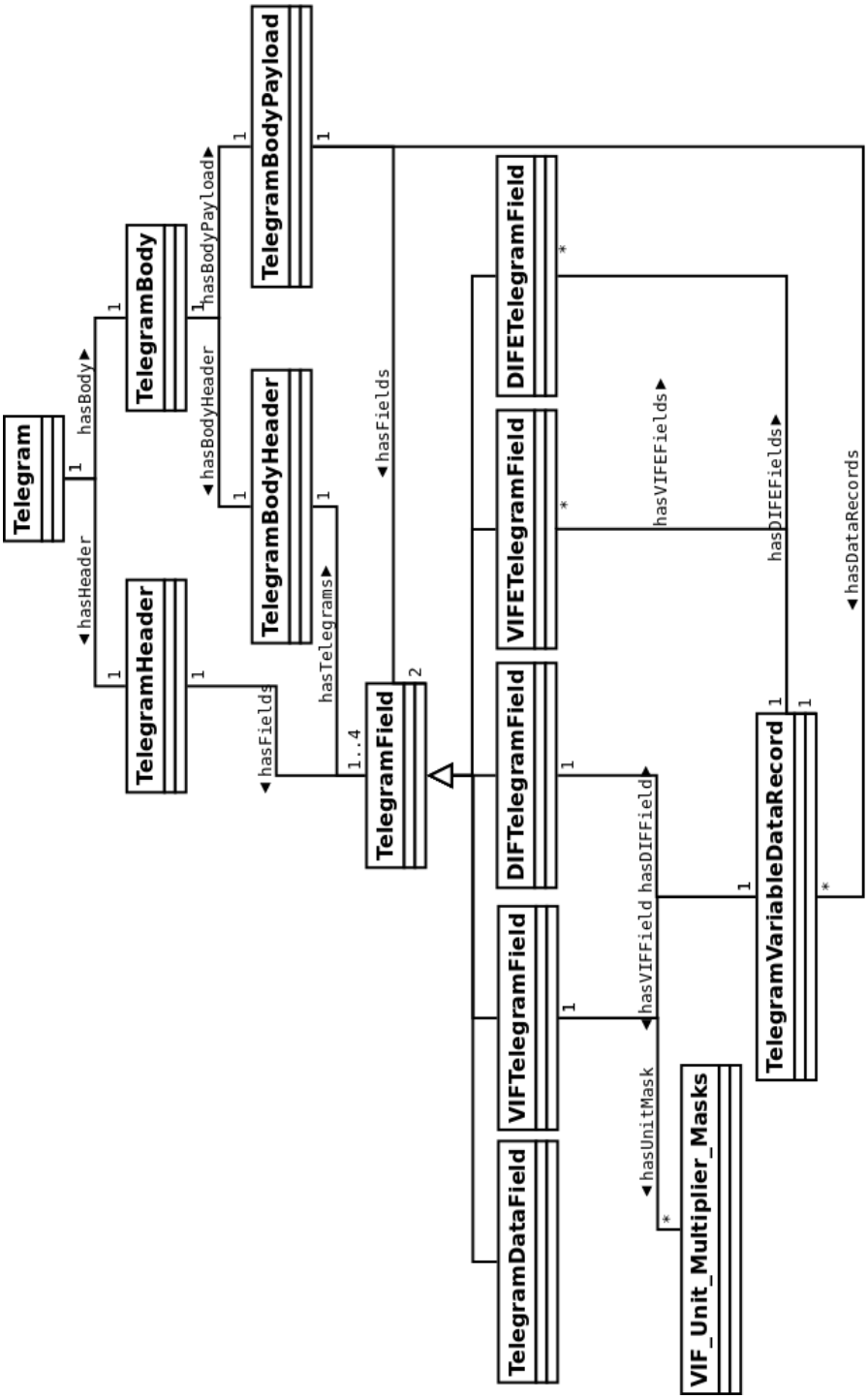

Figure 4.2: Class diagram that provides an overview of the telegram part of the API

In addition to the telegram structure, a manager is necessary to control and store all telegrams. This manager is used to connect to the actual meter device as well as parse telegram values. The manager can be used to observe the connector itself and react to incoming telegrams from the meter device. Figure 4.3 gives an overview.

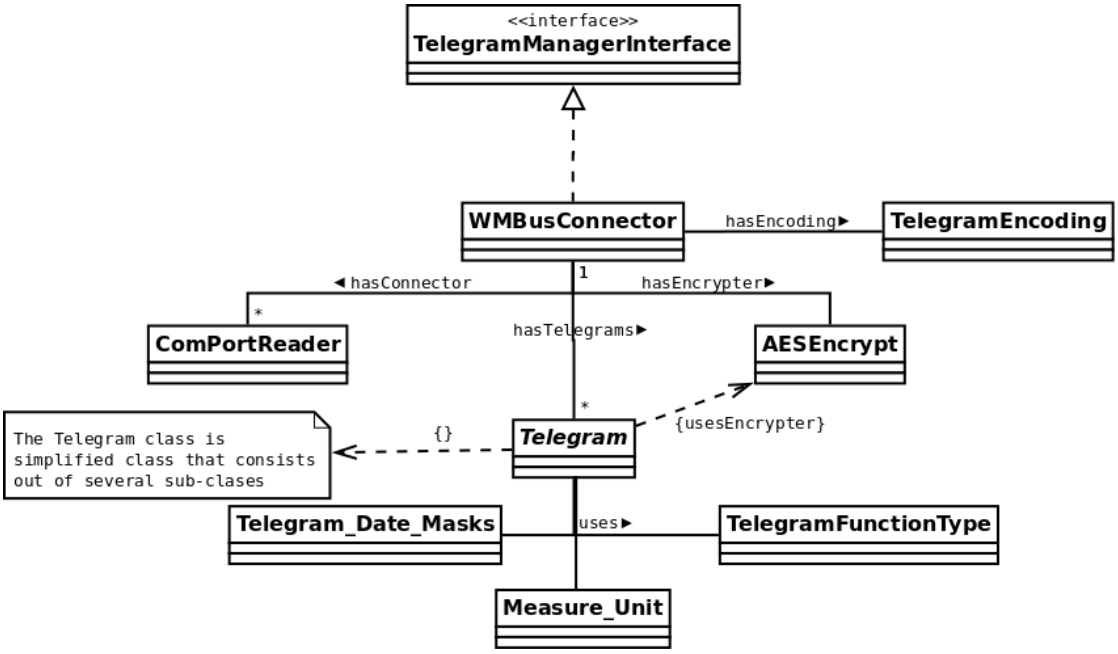

Figure 4.3: Telegram Manager class diagram

To instantiate a telegram manager, the interface *TelegramManagerInterface* has to be implemented. This interfaces already provides the necessary methods to process incoming telegrams. Incoming telegrams are received in the *ComPortReader* class and are then handed over to the class that implements *TelegramManagerInterface* (in this case *WMBusConnector*). The interface enables the creation of custom telegram managers. The telegram manager is also responsible to encrypt and decrypt telegrams as well as parse them as soon as they are decrypted. Both values, raw and parsed, are stored in the telegram manager.

This concludes the overview of the API and the short introduction of its internal structure. As the API is also used and integrated into the IoTSyS framework there is also a description in Section 4.4. Please consider that not all aspects are covered in full detail here since this would go beyond the scope of this paper.

4.2 Telegram structure

In this section, there is an example given for the telegram structure for the configuration of the meter device mentioned above. Please note that other configurations or meter devices can send telegrams with a different structure and although support for various telegram structures has been implemented in the prototype, not all available telegrams might be interpreted erroneous.

The prototype set up was configured to send telegrams with a long header every 60 seconds as a SND_NR telegram. The structure contains first a header and then a body. The body part itself can be split into its header and its payload. The telegram structure is shown in Listing 4.2.

```
Telegram Header:
  L-Field:  1 Byte (length of telegram)
  C-Field:  1 Byte (control field)
  M-Field:  2 Byte (manufacturer field)
  A-Field:  6 Byte (address field)

Telegram Body:
  Header:
    CI-Field: 1 Byte (control input field)
    Acc-Field:  1 Byte (access number field)
    S-Field:  1 Byte (status field)
    Sig-Field:  2 Byte (signature field)
  Body:
    Payload:    value of L-Field minus header length bytes
```

Listing 4.2: prototype meter device telegram structure

The fields described in Listing 4.2 encode various configuration settings in one single field-/byte (for example, the *Sig-Field* encodes information about the encryption as well as the type of communication). Thus, it is necessary to have algorithms in place to extract the information needed and process them accordingly. There is already an overview on these fields given in Section 2 and therefore only an explanation on some fields is provided here.

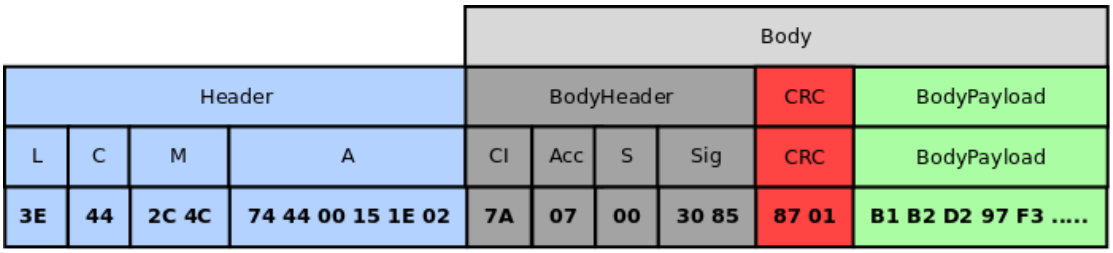

Figure 4.4: An example for a wireless M-Bus telegram

Figure 4.4 shows an example for a telegram and how it is split into several parts (*header*, *body header* and *body payload*). To give a better understanding of how such a telegram is processed this example telegram is analysed in more detail in Listing 4.3.

```
Telegram Header:
  L-Field:  3E    (length of the telegram 62 Byte)
  C-Field:  44    (value 44 means send with no answer)
  M-Field:  2C 4C     (manufacturer converts to "SAM")
  A-Field:  74 44 00 15 1E 02
    SerialNr: 74 44 00 15 (translates to 15004474)
      Version:  1E    (version 30)
      Type: 02    (type 02 electricity)

Telegram Body:
  Header:
    CI-Field: 7A    (7A marks a 4 byte header)
    Acc-Field:  07    (current access number)
    S-Field:  00    (indicating no errors)
    Sig-Field:  30 85 (30 85 indicates an AES encrypted telegram)
  Body:
    Encryption:   87 01 (encryption verification)
    Payload:      (encrypted payload)
    B1 B2 D2 97 F3 7A 9A DB 75 31 11...
```

Listing 4.3: prototype meter device telegram example

Since the telegram shown used in the prototype is quite long only the interesting parts are shown in Listing 4.3. Some of the fields, like for example the *M-Field*, require a more complex calculation to retrieve the correct value. These calculations are omitted here and can be found in [16]. The body payload itself is encrypted and basically consists of several blocks. Each block starts with a *DIF* field and is followed by a *VIF* field. Depending on the value of these two fields the rest of the block is organized. There are multiple options on how such a block can be constructed and therefore only the overall structure is shown here.

As the setup during the prototype implementation used an AES encryption the body payload of the telegram has been encrypted. To encrypt and decrypt telegrams a 16-Byte key along with a CBC (Cipher Block Chaining) initialization vector is required. The method used for the AES encryption might differ and thus no initialization vector is necessary. However, a short example on how such a key is constructed is given in Listing 4.4. Please note that the telegram header itself is not encrypted.

```
AES-Key:
    66 77 66 77 66 77 66 77 66 77 66 77 66 77 66 77

CBC Initialisation vector:
    = M-Field + A-iFeld + 8 Byte-AccessNr
    = 2D 4C 74 44 00 15 1E 02 07 07 07 07 07 07 07 07
```

Listing 4.4: construction of telegram encryption key

Using the setting from Listing 4.4 the telegram can now be decoded.

```
RAW:
    3E 44 2D 4C 74 44 00 15 1E 02 7A 07 00 30 85 87 01
    B1 B2 D2 97 F3 7A 9A DB 75 31 11 25 14 93 FA 8C 4A
    82 CD E1 F2 BB C9 F5 30 E9 A2 3F 1D 2B A7 5D B6 CA
    E4 4A 39 5D 4F 12 E2 12 1E 60 70 43

```

```
Decrypted:
    3E 44 2D 4C 74 44 00 15 1E 02 7A 07 00 30 85 2F 2F
    06 6D 09 87 C0 A1 01 34 04 03 2A ED 10 00 04 83 3C
    00 00 00 00 04 2B 18 00 00 00 04 AB 3C 00 00 00 00
    2F 2F 2F 2F 2F 2F 2F 2F 2F 2F 2F 2F

Parsed:
    44h SAM 15004474 30 02h 7Ah Response from device,
    HL = 4, M-Bus
    4 00h 3085h 1 04h 03h 4 0 0
    Instantaneous value 0 1109289 10^0 Wh Energy
```

Listing 4.5: Example telegram in different states

The examples given in this section only provide a basic overview on how telegrams can be processed. For an expletive description please see Section 2.3 or refer to [16].

4.3 Java Library Usage

To give a better understanding of the Java Library Architecture described in Section 4.1 a short example on how the API can be accessed and used in an application is provided here. The main focus is to show how values of telegrams received by a smart meter device can be retrieved. In addition, a sample implementation for the *TelegramManagerInterface* is given that demonstrates how telegrams can be processed. The implementation of the *TelegramManager* can vary depending on the application. The API provides a *TelegramManager* for the standard use case of receiving and processing smart meter telegrams.

The first part of this section will focus on the *TelegramManager* itself and the latter part will demonstrate how it can be used in an application. Please keep in mind that only a simple example can be shown here because the use cases can be quite complex and be embedded in larger scale applications.

```java
package at.ac.tuwien.auto.iotsys.gateway.connector.wmbus;

import java.util.List;

import at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams.Telegram;

public interface TelegramManagerInterface {

  /**
   * Processes the passed telegram and adds it to its internal storage
   * @param telegramString  String representation of the telegram to be added
   */
  public abstract void addTelegram(String telegramString);

  /**
   * Processes the passed telegram and adds it to its internal storage
   * @param telegram  The telegram to be added
   */
  public abstract void addTelegram(Telegram telegram);

  /**
```

```
22      * Registers the serialNr of the smart meter and the AES key that is used
23      * to encrypt telegrams
24      * @param serialNr  The serialNr of the smart meter
25      * @param aesKey  The AES Key used for the encryption
26      */
27     public void registerAESKey(String serialNr, String aesKey);
28
29     /**
30      * Returns a list of telegrams currently stored
31      * @return  the list of telegrams
32      */
33     public List<Telegram> getTelegrams();
34
35 }
```

Listing 4.6: the interface that telegram manager need to implement

Listing 4.6 shows the interface that each *TelegramManager* has to implement. The implementation can be different depending on the use case and the API provides only a reference implementation. The following listings will only focus on the more interesting parts of the *TelegramManager* and demonstrate how such a *TelegramManager* could operate.

```
1  /**
2   * Processes the passed telegram and adds it to its internal storage
3   * @param telegramString  String representation of the telegram to be added
4   */
5  @Override
6  public void addTelegram(String telegramString) {
7    Telegram telegram = this.createTelegram(telegramString);
8    if(telegram != null) {
9      this.telegrams.add(telegram);
10   }
11 }
12
13 /**
14  * Based on the passed string a telegram is created (if possible). The telegram
15  * is, if necessary, decrypted and parsed. The final telegram is then returned
16  * @param telegramString  String representation of the telegram to be parsed
17  * @return          the created telegram
18  */
19 private Telegram createTelegram(String telegramString) {
20   Telegram telegram = new Telegram();
21   telegram.createTelegram(telegramString, false);
22   if(telegram.decryptTelegram(aesKey) == false) {
23     log.severe("Decryption of AES telegram not possible.");
24     return null;
25   }
26   telegram.parse();
27
28   return telegram;
29 }
```

Listing 4.7: the add method of a TelegramManager

Listing 4.7 shows the methods that are of interest to process a telegram. It is assumed that the *TelegramManager* has an internal storage for incoming telegrams represented as a *java.util.List*.

Furthermore, it is assumed that the necessary set up (including the key to decrypt telegrams) has been provided.

The first method overrides the corresponding method from the *TelegramManagerInterface* and simply calls an internal method that is used to construct a telegram from the passed string. The resulting telegram is then, if not null, added to the internal storage, in this case a *java.util.List* variable, and then the method returns. The second method constructs the telegram itself from a string that represents a telegram. In this case, the construction is rather simple as it is assumed that the variable *telegramString* provides a valid representation of the telegram. In addition, the telegram is then decrypted and parsed right away. The decryption and parsing methods are provided by the *Telegram* class of the API and is automatically only performed if necessary. The resulting telegram is then returned to the caller. Please keep in mind that this just provides a simple example on how such a telegram can be processed. In other use cases, more elaborate checks or steps may be necessary.

Now that the functionality of the *TelegramManager* has been established, an overview on how to use the API and in particular this *TelegramManager* can be provided. For a better understanding, first a step by step description is given and in conclusion the full listing is shown. The goal is to receive smart meter telegrams and extract power and energy values out of them.

```
1 // first the telegram manager is initialized
2 // and the corresponding serial number and AES key (if necessary) is set
3 TelegramManagerInterface tManager = new TelegramManager();
4 tManager.registerAESKey(SmartMeterStarter.SERIAL_NR, SmartMeterStarter.AES_KEY);
```

Listing 4.8: initializing the TelegramManager

The first step is to initialize the *TelegramManager* and use the correct serial numbers and AES key for its set up. Listing 4.8 shows how such a set up is accomplished.

```
1 // find the COM-Port under which the wireless M-Bus USB stick can be accessed
2 CommPortIdentifier portId = ComPortReader.lookupPorts("/dev/ttyUSB0");
3 // and initialize the connector with these values
4 ComPortReader reader = new ComPortReader(portId, tManager);
```

Listing 4.9: initializing the ComPortReader

The second step is to initialize the connector *ComPortReader*. The *ComPortReader* is used to establish a connection through a COM-Port (in this set up a wireless M-Bus USB stick has been used). First, the correct port is identified by providing a device URI and a lookup and afterwards the *ComPortReader* itself can be initialized. While initializing the *ComPortReader* receives a reference to the previously generated *TelegramManagerInterface* stored in the variable *tManager*. This is necessary so that the *ComPortReader* can notify the *TelegramManagerInterface* via its *addTelegram* method. As soon as the *ComPortReader* is initialized it listens to incoming telegrams and, if it receives one, passes it to the *TelegramManagerInterface*.

After the *ComPortReader* has been initialized the set up is done and the API is already receiving incoming telegrams and parsing them. Everything that is now missing is to extract some

values of the incoming telegrams. Since this is a simplified example it is assumed that, after initialization, the *TelegramManager* immediately receives some telegrams. Usually, the application logic would be placed between the initialization and the data extraction, or even more often the *TelegramManager* itself would be used directly in the application logic.

```
1 // retrieve a list of currently stored telegrams
2 List<Telegram> telegrams = tManager.getTelegrams();
3
4 if(telegrams != null) {
5   for(Telegram telegram : telegrams) {
6     System.out.println("Energy Value: " + telegram.getEnergyValue());
7     System.out.println("Power Value: " + telegram.getPowerValue());
8   }
9 }
```

Listing 4.10: retreiving telegrams from the TelegramManager

Listing 4.10 shows how data can be extracted from telegrams. First, a list of telegrams is received from the *TelegramManager* via the *getTelegrams* method. Afterwards, each telegram gets processed in a loop and the values are easily extracted by the *getEnergyValue* and *getPowerValue* methods of the API.

```
1 // finally close the port which is connected to the smart meter
2 reader.closePort();
```

Listing 4.11: closing the connection of the ComPortReader

To close the connection to the smart meter, the open connection in the *ComPortReader* has simply to be closed. Listing 4.11 shows how this is accomplished.

The code fragments from the previous listings can be combined to form a simple application and is showing as a full code segment in Listing 4.12.

```
1  package at.ac.tuwien.auto.iotsys.gateway.connector.wmbus;
2
3  import gnu.io.CommPortIdentifier;
4
5  import java.util.List;
6
7  import at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.reader.ComPortReader;
8  import at.ac.tuwien.auto.iotsys.gateway.connector.wmbus.telegrams.Telegram;
9
10 public class SmartMeterStarter {
11   /**
12    * serial number of the smart meter
13    */
14   private static String SERIAL_NR = "15004474";
15
16   /**
17    * the AES key that is used to encrypt meter telegrams
18    */
19   private static String AES_KEY = "66 77 66 77 66 77 66 77 66 77 66 77 66 77 66 77";
20
```

```
21    public static void main(String[] args) {
22      // first the telegram manager is initialized
23      // and the corresponding serial number and AES key (if necessary) is set
24        TelegramManagerInterface tManager = new TelegramManager();
25        tManager.registerAESKey(SmartMeterStarter.SERIAL_NR, SmartMeterStarter.AES_KEY)↩
              ;
26
27        // find the COM-Port under which the wireless M-Bus USB stick can be accessed
28        CommPortIdentifier portId = ComPortReader.lookupPorts("/dev/ttyUSB0");
29        // and initialize the connector with these values
30        ComPortReader reader = new ComPortReader(portId, tManager);
31
32        // retrieve a list of currently stored telegrams
33        List<Telegram> telegrams = tManager.getTelegrams();
34
35        if(telegrams != null) {
36          for(Telegram telegram : telegrams) {
37            System.out.println("Energy Value: " + telegram.getEnergyValue());
38            System.out.println("Power Value: " + telegram.getPowerValue());
39          }
40        }
41
42        // finally close the port which is connected to the smart meter
43        reader.closePort();
44    }
45
46  }
```

Listing 4.12: full code example for a smart meter reader class

4.4 Integration into oBIX based IoTSyS framework

In addition to the standalone prototype described in Section 4.1, a plugin for a the IoTSyS framework has been developed. IoTSyS aims at providing an integration middleware for the Internet of Things and allows interoperability between different devices. IoTSyS makes use of OSGi (Open Services Gateway initiative) and thus an approach for modularization via standard APIs has been implemented. This means that single connectors can be bundled and integrated into the framework without changes on the actual source code. The communication stack includes embedded devices based on IPv6, Web Services and the oBIX standard to enable simple and efficient information exchange in a smart grid environment [26].

The basic idea was to include the prototype implementation for the smart meter reader as a plugin into the existing framework. To accomplish this task, a couple of new classes has been introduced to the prototype. IoTSyS provides a web server that allows to access registered entities throughout a common interface - oBIX. Figure 4.5 gives an overview on how the communication flow is realized.

The interaction with the IoTSyS framework is already built into the Java library and connectors and device loaders have been put into place. Figure 4.6 shows a class diagram for this matter. Please note that the *wMBusConnector* class is only an abstract class which stands for a more complex build up described in Section 4.1. The *SmartMeter* interface provides the mapping for

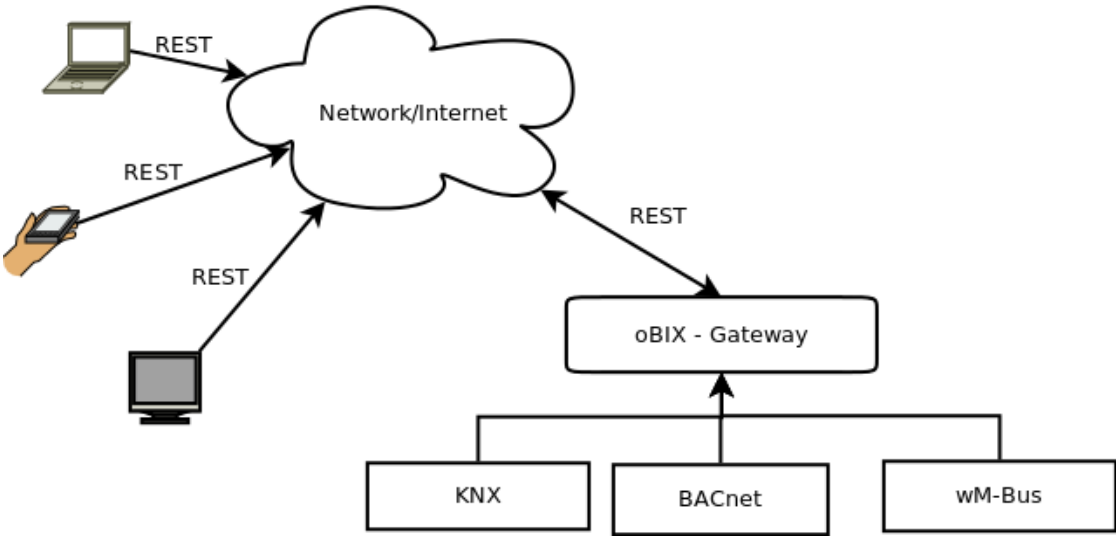

Figure 4.5: Schematic overview of oBIX framework with wMBus plugin

oBIX objects and is specified according to the defined contract. The *WMBusBundleActivator* is used as an entry point from the IotSyS framework and activates the device implementation through which the actual *WMBusConnector* is then initialized. The *WMBusWatchDog* interface allows to register a watch dog which is used to notify about new incoming data values.

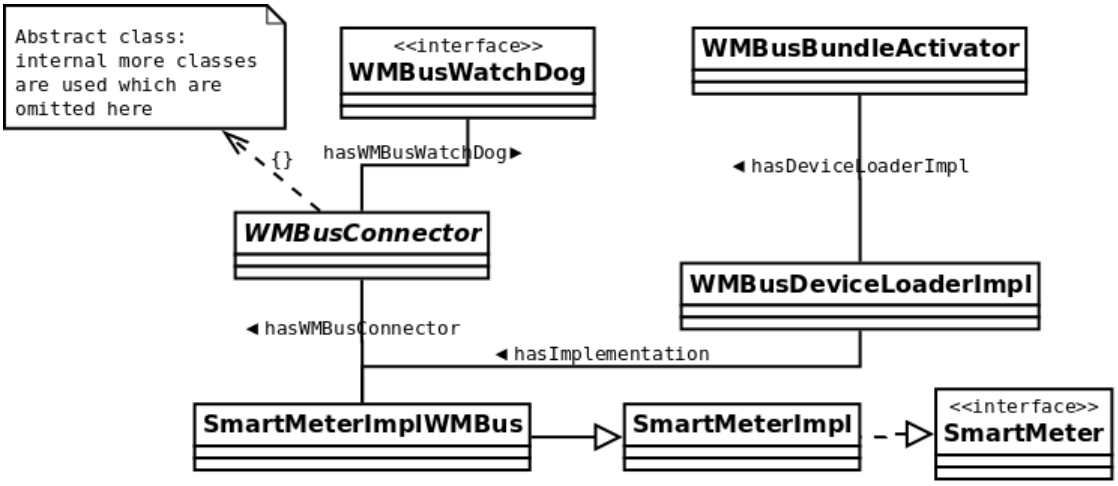

Figure 4.6: Class diagram of WMBusConnector and IoTSyS framework

Since all registered entities operate according to the oBIX standard interoperability is achieved. A list of registered devices can be accessed by calling the start URL of the web server (in this case `http://localhost:8080`). To discuss the integration of the API into the IoTSyS framework it is necessary to revisit Listing 3.4 from Section 3.3 (for convenience the listing is, slightly simplified, included here as well and can be seen in Listing 4.13).

```
<obj name="history" href="/smartMeter/energy/history">
  <int name="count" href="count" val="50"/>
  <abstime name="start" href="start" val="2013-05-22T21:43:30.298+02:00" tz="Europe/↩
      Vienna"/>
  <abstime name="end" href="end" val="2013-05-22T21:48:27.355+02:00" tz="Europe/↩
      Vienna"/>
  <op name="query" in="obix:HistoryFilter" out="obix:HistoryQueryOut"/>
</obj>
```

Listing 4.13: Simplified oBIX history object

The operation specified in the history contract can be invoked via a POST request.

```
<obj name="query" is="obix:HistoryFilter">
  <int name="limit" val="5" />
  <abstime name="start" null="true" />
  <abstime name="end" null="true" />
</obj>
```

Listing 4.14: query for an oBIX history object

To invoke such an operation, a POST request is necessary. The content of this request is another oBIX object with a specific contract (for the query operation we have to use the *obix:HistoryFilter* contract). The history object is queried via the URL `http://localhost:8080/smartMeter/energy/history`. Listing 4.14 shows the content of such a query POST request. The attribute *name* specifies which operation is to be invoked and the following possible values (*limit*, *start* and *end*) are specified in the contract used (*obix:HistoryFilter*). In this case, there is no time constraint given but the number of results is limited to five.

```
<obj is="obix:HistoryQueryOut">
  <int name="count" href="count" val="5"/>
  <abstime name="start" href="start" val="2013-05-22T22:08:15.602+02:00" tz="Europe/↩
      Vienna"/>
  <abstime name="end" href="end" val="2013-05-22T22:08:27.605+02:00" tz="Europe/↩
      Vienna"/>
  <list of="obix:HistoryRecord">
    <obj>
      <abstime val="2013-05-22T22:08:15.602+02:00" tz="Europe/Vienna"/>
      <real val="1.0380416666666723"/>
    </obj>
    <obj>
      <abstime val="2013-05-22T22:08:18.603+02:00" tz="Europe/Vienna"/>
      <real val="1.0388750000000055"/>
    </obj>
    <obj>
      <abstime val="2013-05-22T22:08:21.603+02:00" tz="Europe/Vienna"/>
      <real val="1.0397083333333388"/>
```

```
17      </obj>
18      <obj>
19        <abstime val="2013-05-22T22:08:24.604+02:00" tz="Europe/Vienna"/>
20        <real val="1.040541666666672"/>
21      </obj>
22      <obj>
23        <abstime val="2013-05-22T22:08:27.605+02:00" tz="Europe/Vienna"/>
24        <real val="1.0413750000000053"/>
25      </obj>
26    </list>
27  </obj>
```

Listing 4.15: result for a queried oBIX history object

Listing 4.15 shows the result for the query in Listing 4.14. The count attribute indicates that are only five values contained in the result set (which is according to the contract *obix:HistoryQueryOut*). The results themselves can be found in a contract of *obix:HistoryRecord* which holds multiple values (in this case five). This listing also shows an example for nested objects in oBIX.

In addition, it is possible to register at a *watchService* and get value changes in near real-time. An optional history object for the smart meter implementation is provided as well. This *watchService* allows the monitoring of meter devices and provides a near-real time reporting functionality on energy consumption. Utilizing this feature it is possible to meet the legal ordinance issued by the Austrian Government (IMA-VO and IME-VO).

For a full list of features please refer to [26].

Evaluation

In the overall setup there are two parts of interest. First the actual meter device which is a Siemens AMIS Smart Meter with an additional Amber wireless M-Bus module which enables communication via the wireless M-Bus standard. The configuration of the meter can be seen in Listing 4.16. This device operates as slave during the interaction. Second the receiver itself which is an Amber AMB8425-M wireless M-Bus USB stick which simulates a virtual COM-Port that allows a transparent access similar to normal M-Bus devices. In this setup the USB stick is used as the master and is connected to a device that is used to receive or interact with the meter itself. Basically this stick can be used on each device that provides the necessary drivers.

```
1 Smart Meter:
2   ID:   15007774
3   MAN:  SAM
4   Type:  02 (Electricity)
5   Version: 30
```

Listing 4.16: smart meter configuration

The meter has been configured to send its values in a periodic interval (every 60 seconds) as an SND_NR telegram.

The overall set up has been tested on a Windows 7 and a Linux installation.

CHAPTER 5

Conclusion

Modern technologies facilitate the usage of various devices to control and administrate systems remotely. Consumers wish to be able to have access to data everywhere and at any time. Ubiquitous computing is a common term and describes the need of interoperability between different kinds of devices, often also referred to the Internet of Things. These technologies provide us with the chance to build complex and easy to use home automation systems that can be part of a smart grid infrastructure. Smart meter devices are the foundation of this infrastructure and can be used to access and supervise devices in home automation systems. However, not all of the questions surrounding this task have yet been solved.

This bachelor thesis therefore presents a JAVA library for the M-Bus standard that provides an easy to use and extendible API. Building upon this prototype, higher level applications can be built that utilize the M-Bus communication protocol without the need of all the lower level complexity of it. The prototype implemented provides a configurable setup and can be used on various platforms. In addition, it allows encrypted telegrams to be processed so that data exchange can be realized in a secure way. The approach used has shown that such an abstract API can be realized and that M-Bus telegrams can be processed in near real-time. This enables the consumer to have accurate and up-to-date information about devices connected to a smart meter. Further implementations can provide control mechanisms with a UI so that the information can be visualized in a more graphic way.

Furthermore, the prototype has been integrated into an oBIX server where the meter values can be accessed through a common Web Service infrastructure. This supports the interoperability of the device through a common standardized communication protocol - the oBIX standard. oBIX proposes an XML based Web Service infrastructure and allows access to devices through simple Web Service calls. The IoTSyS framework makes use of the oBIX standard and supplies standardized OSGi bundles that can be integrated. By doing so they can be accessed through a Web Service and communicate via the oBIX standard and thus are able to accomplish machine-to-machine interaction and moreover automatically process information in between them.

List of Figures

2.1 M-Bus Standard Setup 4
2.2 Physical Layer - Bit representation: Page 6 of [15] 8
2.3 Wireless-M-Bus communication - Simplified bidirectional communication (as in T2) 13

4.1 Package diagram that provides a schematic overview of the API 22
4.2 Class diagram that provides an overview of the telegram part of the API 24
4.3 Telegram Manager class diagram 25
4.4 An example for a wireless M-Bus telegram 26
4.5 Schematic overview of oBIX framework with wMBus plugin 33
4.6 Class diagram of WMBusConnector and IoTSyS framework 33

List of Tables

2.1 M-Bus Protocol Stack 7
2.2 Data Link Layer - telegram formats M-Bus 9

Listings

3.1 oBIX object SmartMeter which holds an endpoint 19

3.2 complex oBIX object 19
3.3 oBIX writeable object 19
3.4 oBIX history object 20
4.1 java prototype package structure 23
4.2 prototype meter device telegram structure 26
4.3 prototype meter device telegram example 27
4.4 construction of telegram encryption key 27
4.5 Example telegram in different states 27
4.6 the interface that telegram manager need to implement 28
4.7 the add method of a TelegramManager 29
4.8 initializing the TelegramManager 30
4.9 initializing the ComPortReader 30
4.10 retreiving telegrams from the TelegramManager 31
4.11 closing the connection of the ComPortReader 31
4.12 full code example for a smart meter reader class 31
4.13 Simplified oBIX history object 34
4.14 query for an oBIX history object 34
4.15 result for a queried oBIX history object 34
4.16 smart meter configuration 36

Bibliography

[1] RXTX - SerialPort Interface. http://rxtx.qbang.org/wiki/index.php/Main_Page. [Online; accessed 03-March-2013].

[2] ANSI - American National Standards Institute. Protocol Specification for ANSI Type 2 Optical Port C12.18, 2006.

[3] ANSI - American National Standards Institute. Protocol Specification for Interfacing to Data Communication Networks C12.22, 2008.

[4] Carsten Bories. Einrichtung einer intelligenten Ausleseeinheit für Verbrauchsmesszähler. Master's thesis, Fachbereich Physik Universität - GH Paderborn, 1995.

[5] P. Bredillet, E. Lambert, and E. Schultz. Cim, 61850, COSEM Standards Used in a Model Driven Integration Approach to Build the Smart Grid Service Oriented Architecture. In *2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 467–471, 2010.

[6] Bundeskanzleramt Österreich. Intelligente Messgeräte - Anforderungs VO - IMA-VO 2011. http://www.ris.bka.gv.at/GeltendeFassung/Bundesnormen/20007497/IMA-VO%202011%2c%20Fassung%20vom%2012.07.2013.pdf. [Online; accessed 24-June-2013].

[7] BUNDESMINISTERIUM für WIRTSCHAFT, FAMILIE und JUGEND. Aktuelle Rechtsvorschriften - Österreich. http://www.bmwfj.gv.at/ministerium/rechtsvorschriften/kundgemachte_rechtsvorschriften/seiten/listeaktuellerrechtsvorschriftenab112009.aspx. [Online; accessed 24-June-2013].

[8] BUNDESMINISTERIUM für WIRTSCHAFT, FAMILIE und JUGEND. Intelligente Messgeräte - Einführungsverordnung IME-VO. http://www.bmwfj.gv.at/Ministerium/Rechtsvorschriften/kundgemachte_rechtsvorschriften/Documents/Intelligente%20Messger%C3%A4te.pdf. [Online; accessed 24-June-2013].

[9] Christof Hoentzsch. The M-Bus: A Documentation Rev. 4.8. http://www.m-bus.com/mbusdoc/md4.php. [Online; accessed 24-June-2013].

[10] L. Coetzee and J. Eksteen. The Internet of Things - promise for the future? An introduction. In *Proceedings of IST-Africa Conference 2011*, pages 1–9, 2011.

[11] DIN Deutsches Institut für Normung e. V. Lokales Bussystem, DIN EN 13757-6, 2003.

[12] DIN Deutsches Institut für Normung e. V. Weitervermittlung, DIN EN 13757-5, 2003.

[13] DIN Deutsches Institut für Normung e. V. Zählerauslesung über Funk (Fernablesung von Zählern im SRD-Band), DIN EN 13757-4, 2003.

[14] DIN Deutsches Institut für Normung e. V. Kommunikationssysteme für Zähler und deren Fernablesung, DIN EN 13757-1, 2005.

[15] DIN Deutsches Institut für Normung e. V. Physical und Link Layer, DIN EN 13757-2, 2005.

[16] DIN Deutsches Institut für Normung e. V. Spezielle Anwendungsschicht, DIN EN 13757-3, 2005.

[17] S. Feuerhahn, M. Zillgith, C. Wittwer, and C. Wietfeld. Comparison of the communication protocols DLMS/COSEM, SML and IEC 61850 for smart metering applications. In *Smart Grid Communications (SmartGridComm), 2011 IEEE International Conference on*, pages 410–415, 2011.

[18] A. Flammini, S. Rinaldi, and A. Vezzoli. The sense of time in open metering system. In *Smart Measurements for Future Grids (SMFG), 2011 IEEE International Conference on*, pages 22–27, 2011.

[19] Bin Guo, Daqing Zhang, and Zhu Wang. Living with Internet of Things: The Emergence of Embedded Intelligence. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 297–304, 2011.

[20] Wolfgang Hascher. Wireless-M-Bus – der neue Smart-Metering-Standard? `http://www.elektroniknet.de/kommunikation/technik-know-how/kommunikations-module-u-systeme/article/1530/0/Wireless-M-Bus__der_neue_Smart-Metering-Standard/`, 2009. [Online; accessed 10-March-2013].

[21] Horst Ziegler, Carsten Bories. M-bus: Ausdehnung des Netzes bei unterschiedlichen Baudraten. *Fachbereich Physik Universität - GH Paderborn*, dec 1995.

[22] International Electrotechnical Commission. Electricity metering - Data exchange for meter reading, 2002.

[23] H. Jarvinen, A. Litvinov, and P. Vuorimaa. Integration platform for home and building automation systems. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 292–296, 2011.

[24] Wang Jiahui, Liu Xiaodan, Zeng Lei, and Hou Weiyan. The design and implementation of a wireless meter reading system. In *2011 10th International Conference on Electronic Measurement Instruments (ICEMI)*, volume 1, pages 115–120, 2011.

[25] I. Kunold, M. Kuller, J. Bauer, and N. Karaoglan. A system concept of an energy information system in flats using wireless technologies and smart metering devices. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, volume 2, pages 812–816, 2011.

[26] Markus Jung. IoTSyS - Internet of Things integration middleware. `http://code.google.com/p/iotsys`. [Online; accessed 22-March-2013].

[27] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner. Web Services in Building Automation: Mapping KNX to oBIX. In *2007 5th IEEE International Conference on Industrial Informatics*, volume 1, pages 87–92, 2007.

[28] OASIS Open Building Information Exchange TC. OBIX Version 1.1, Working Draft 06, 2010.

[29] M.M. Rahman and A. Mto. Technologies required for efficient operation of a smart meter network. In *Industrial Electronics and Applications (ICIEA), 2011 6th IEEE Conference on*, pages 809–814, 2011.

[30] C. Selvam, K. Srinivas, G. S. Ayyappan, and M. Venkatachala Sarma. Advanced metering infrastructure for smart grid applications. In *Recent Trends In Information Technology (ICRTIT), 2012 International Conference on*, pages 145–150, 2012.

[31] Siemens AG. Smart Metering Infrastructure. `http://www.energy.siemens.com/co/en/energy-topics/smart-grid/smart-consumption/smart-metering-infrastructure.htm`. [Online; accessed 24-November-2011].

[32] Steinbeis Transfer Center - Embedded Design and Networking. Wireless M-Bus Documentation. `http://www.stzedn.de/wireless-m-bus-stack.html`. [Online; accessed 24-April-2012].

[33] Hou Weiyan, Wang Jiahui, and Zhang Fangchang. A scheme for the application of smart message language in a wireless meter reading system. In *2011 Third International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, volume 1, pages 254–257, 2011.

[34] Ling Zheng, Shuangbao Chen, Shuyue Xiang, and Yanxiang Hu. Research of architecture and application of Internet of Things for Smart Grid. In *2012 International Conference on Computer Science Service System (CSSS)*, pages 938–941, 2012.