Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Christofer Gilje Skjæveland

Infrastructure for Collecting and Analysing near Real-Time Data from Several Water Meters Using Wireless M-Bus

Infrastruktur for innsamling og analysering av nær sanntidsdata fra en rekke vannmålere, ved bruk av trådløs M-Bus

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen
May 2021

NTNU
Norwegian University of
Science and Technology

Christofer Gilje Skjæveland

Infrastructure for Collecting and Analysing near Real-Time Data from Several Water Meters Using Wireless M-Bus

Infrastruktur for innsamling og analysering av nær sanntidsdata fra en rekke vannmålere, ved bruk av trådløs M-Bus

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology
and Electrical Engineering
Department of Engineering Cybernetics

MASTER THESIS DESCRIPTION

Candidate:	**Christofer Gilje Skjæveland**
Course:	**TTK4900 Engineering Cybernetics**
Thesis title (Norwegian):	**Infrastruktur for innsamling og analysering av nær sanntidsdata fra en rekke vannmålere, ved bruk av trådløs M-Bus**
Thesis title (English):	**Infrastructure for Collecting and Analysing near Real-Time Data from Several Water Meters Using Wireless M-Bus**

Thesis description: By having household water meters equipped with near real-time flow and pressure measurements and the readout of several meters are done near simultaneously, it may be possible to analyze the condition of the water grid that connects the households.

In this thesis, we want to develop an infrastructure for wireless collection of near real-time flow and pressure measurements from several households in a neighbourhood. The work should include indications on how the measurements could be used for indicating leakages in the water grid outside the buildings.

The tasks will be:

1. Conduct a literary study of systems used for wireless collection of measurements from household water meters. Also, perform a study of how such measurements can be used for leakage detection/localization.
2. Propose an infrastructure system for collecting near real-time flow and pressure measurements from several households in a neighbourhood. The proposal shall indicate how algorithms for leakage detections can be included in the system.
3. As far as time permits, implement the suggested system.

Start date: January 4th, 2021
Due date: May 31th, 2021

Thesis performed at: Department of Engineering Cybernetics
Supervisor: Professor Geir Mathisen, Dept. of Eng. Cybernetics

Foreword

The work presented in this thesis was conducted for the Department of Engineering Cybernetics, at the Norwegian University of Science and Technology (NTNU).

The work was performed under the supervision of Professor Geir Mathisen, from the Department of Engineering Cybernetics. Geir has provided equipment to be used, report structure, technical insight and general guidance throughout the process.

This thesis is a continuation of work performed in a specialization project by Skjæveland C. G. [1]

Abstract

The use of Smart water meters are on the rise, and several municipals in Norway, including Trondheim, wants every household to implement it. Wireless Meter-Bus (wM-Bus) has been shown to be a popular standard in this process and it gives an opportunity to develop a new infrastructure to collect all data near real-time. In addition, by adding pressure meters to work alongside, this might give more insight of the water grid connecting the households and therefore give indications of leakages in the grid.

In this project, a infrastructure capable of storing real-time flow and pressure measurements in a cloud solution from several households in a neighborhood was created. This data was then made easily accessible through the use of an Application Programming Interface (API) and also by using a third-party application. The data was further used to create a representational model of the water grid, which was used to compare with real measurements to give indications of leakages.

It is concluded that although the infrastructure works as intended, changes need to be made to make it viable in a real setting. The analysis performed on the data showed that the uncertainty of the model was too great to give any indications of leakages. More knowledge surrounding the water grid is therefore needed improve the model, and therefore the leakage detection and localization strategy.

Sammendrag

Bruken av smarte vannmålere blir mer populært, og flere norske kommuner, inkludert Trondheim, ønker at det tas i bruk for alle husstander. Wireless Meter-Bus (wM-Bus) har vist seg å være en populær standard i denne prosessen og det gir en mulighet til å utvikle en ny insfrastruktur for å samle data i nær sanntid. Ved å i tillegg ha trykkmålere ved siden av så kan det være mulig å få mer innsyn i vannettet som kobler sammen husstandene, og derfor gi en indikasjon på lekasjer.

I dette prosjektet ble det utviklet en infrastruktur for innsamling av vannforbruk- og trykkmålinger i nær sanntid, som lagres i en skyløsning for flere husstander i et nabolag. Denne dataen ble dermed gjort mer tilgjengelig ved bruk av en Application Programming Interface (API) i tillegg til en tredjeparts programvare. Dataen ble videre brukt til å lage en modell som representerer vannettet. Denne modellen kunne da brukes til å generere trykkresidualer fra en sammenligning med ekte måleverdier, noe som er essensielt for indikasjoner av lekasjer.

Det konkluderes med at infrastrukturen fungerer som ønsket, men endringer må gjøres for at den skal fungere i et ekte scenario. Dataanalysen viste at usikkerheten i modellen var for stor for å gi indikasjoner for lekasjer. Mer informasjon om vannettet må dermed brukes for å forbedre modellen, og dermed lekasjedeteksjons- og lekasjelokaliseringsstrategien.

Contents

Foreword . . . . . **i**
Abstract . . . . . **ii**
Sammendrag . . . . . **iii**
Contents . . . . . **iv**
Figures . . . . . **vi**
Tables . . . . . **vii**
Code Listings . . . . . **viii**
Acronyms . . . . . **ix**
1 Introduction **1**
1.1 Background and Motivation 1
1.2 Limitations 2
1.3 Disposition of the Task 2
2 Literary Study **4**
2.1 Communication Technologies for Water Distribution Networks (WDNs) 4
2.2 Smart Water Grid Projects 5
2.3 A Review of Leakage Detection Strategies 6
2.3.1 Classification of Leak Detection Strategies 6
2.3.2 Mass/Volume Balance 8
2.3.3 Negative Pressure Wave (NPW) Method 8
2.3.4 Gradient Intersection Method 9
2.3.5 Using Pressure Residuals From a Model-Based Approach 9
2.3.6 Mixed Model-Based/Data-Driven Approach 9
2.4 Summary 10
3 Theory **12**
3.1 The Wireless M-Bus Standard 12
3.1.1 Physical Layer (PHY) 12
3.1.2 Data Link Layer (DLL) 14
3.1.3 Extended Link Layer (ELL) 16
3.1.4 Network Layer (NWL) 16
3.1.5 Authentication and Fragmentation Layer (AFL) 17
3.1.6 Transport Layer (TPL) 17
3.1.7 Application Layer (APL) 17
3.2 The Open Metering System Standard 19
3.3 The STACKFORCE Protocol Stack 19
3.4 EPANET and water modelling 20
3.4.1 The EPANET Model 20
3.4.2 Conservation of Energy and Mass 21

3.4.3 Analysis Algorithm to Find Total Head and Flow . . . 23
3.4.4 Modelling Water Demand Patterns . . . 23
3.5 Hydraulic Properties of a Leak . . . 24
3.6 Generation of Pressure Residuals . . . 24
4 Design and Specification . . . 26
4.1 Functional Specification . . . 26
4.2 System Overview . . . 26
5 Implementing the Infrastructure . . . 29
5.1 Configuring Simulated Meters SLWSTK6220A . . . 29
5.2 Setting Up the Device Connectivity Layer . . . 32
5.2.1 WM-Bus Collector . . . 32
5.2.2 Serial Port Logger . . . 36
5.3 The Cloud Solution . . . 37
5.3.1 Identity and Access Management (IAM) . . . 37
5.3.2 Cloudwatch . . . 37
5.3.3 IoT Core and Rules . . . 38
5.3.4 Timestream . . . 38
5.3.5 API Gateway and Lambda . . . 40
5.4 Creating the EPANET Model . . . 43
5.4.1 Creating the Demand Pattern . . . 45
5.5 Data Visualization and Analysis . . . 47
5.5.1 Visualization Using Grafana . . . 47
5.5.2 Generating Pressure Residuals . . . 48
6 Testing and Results . . . 49
6.1 Receiving Correct Data and formatting Packets in the Device Connectivity Layer 49
6.2 Testing Cloud Solution Components . . . 50
6.2.1 Receiving data in IoT Core . . . 50
6.2.2 Test Query of Timestream . . . 51
6.2.3 Test of GET Method for API Gateway . . . 51
6.3 Visualization of Data using Grafana . . . 54
6.4 Generated Pressure Residuals . . . 54
7 Discussion . . . 58
7.1 Review of Specifications . . . 58
7.2 Other Improvements . . . 60
8 Conclusion . . . 61
9 Further Work . . . 62
9.1 Further Work for Infrastructure . . . 62
9.2 Further Work for Data Analysis . . . 63
Bibliography . . . 64
A M-Bus Packets . . . 66
B Serial Port Logging Code . . . 69
C CSV to IBM hex format converter . . . 76
D Data Analysis . . . 78
E Simulated Meter Code . . . 81

Figures

2.1 Communication technologies for smart water grid 6
2.2 Five layered architecture for water management. 7
2.3 Classification of major leak detection techniques. 7
2.4 Gradient intersect method. 9
2.5 Leak localization scheme based on a hybrid approach. 10

3.1 Architecture of the STACKFORCE wM-Bus Protocol Stack. 20
3.2 Overview of vocabulary concerning the model. 21
3.3 Total head along a pipeline. 22

4.1 An overview of the whole infrastructure. 28

5.1 The simulated meters used in the project. 33
5.2 Block diagram of the RC1180 demonstration kit. 34
5.3 Overview of MBUS-CCT program window 34
5.4 Settings used for action associated with the AWS IoT rule. 39
5.5 Amazon Timestream architecture overview. 41
5.6 Amazon Timestream storage overview. 41
5.7 Map of the nearby water grid of the water meters. 43
5.8 EPANET model of the water grid. 44
5.9 Water demand pattern. 46
5.10 Water demand pattern in EPANET. 46

6.1 Resulting raw wM-Bus packets. 49
6.2 Resulting formatted wM-Bus packets. 50
6.3 Resulting formatted wM-Bus packets graphed. 51
6.4 Resulting JSON packets being received by IoT Core 52
6.5 Result of query for Timestream 52
6.6 Result of GET API response. 53
6.7 Grafana panel showing the latest packets that have been registered. 54
6.8 Grafana panel showing data for all locations. 55
6.9 Grafana panel showing RSSI measurements from all meters for both collectors. . 56
6.10 Real pressure measurements plotted alongside modelled pressure estimations. . 56
6.11 Generated pressure residuals. 57

A.1 SLWSTK6220A wM-Bus packets used in this project. 67
A.2 Kamstrup M-Bus packets used in this project. 68

Tables

2.1 LPWAN technologies 5

3.1 The wM-Bus protocol as a layer model 13
3.2 wM-Bus operating modes 13
3.3 M-Bus frame format A. 14
3.4 M-Bus frame format B. 14
3.5 C-field data format. 15
3.6 Manufacturer ID's used in this project. 15
3.7 Device types used in this project. 15
3.8 CI-fields used for this project. 16
3.9 Full and compact M-bus frame. 18
3.10 The data information for data records in this project. 18
3.11 The value information for data records in this project. 18

5.1 Identification for every meter used in this project. 32
5.2 Configurations for collector 36
5.3 Values for measure_name. 40
5.4 Parameters for EPANET. 44
5.5 Modifications to Timestream record. 47

Code Listings

5.1 Code modification for simulated flow meters 31
5.2 Terminal commands for collector 35
5.3 Lambda function for API Gateway. 42
5.4 Standard query used for Grafana. 47
B.1 Serial port logging code 69
C.1 CSV to IBM hex formatter 76
D.1 Script for analyzing data 78
E.1 Main code file used for programming simulated meters. 81

Acronyms

AFL Authentication and Fragmentation Layer. 12, 15–17, 19
API Application Programming Interface. ii, iii, vi, viii, 6, 19, 27, 38, 40, 42, 43, 51, 53, 58, 62
APL Application Layer. 12, 14, 15, 17, 19, 20, 29
AWS Amazon Web Services. 6, 27, 28, 37, 38, 40, 42, 60

CI Control Information. 12, 16, 49
COM Communication Port. 33
CRC Cyclic Redundancy Check. 16, 19
CSV Comma-separated values. 36, 37

DIB Data Information Block. 18
DIF Data Information Field. 18
DIFE Data Information Field Extension. 18
DK Demonstration Kit. 33
DLL Data Link Layer. 12, 14, 16, 17, 19
DMA District Metered Area. 1, 2, 8–11, 20, 21, 23–25, 59, 61, 63

ELL Extended Link Layer. 12, 15, 16, 18, 19

GA Gradient Algorithm. 23
GSM Groupe Spécial Mobile. 5

HAL Hardware Abstraction layer. 19, 20
HTTP Hypertext Transfer Protocol. 40

IAM Identity and Access Management. 27, 37, 38, 47
IoT Internet of Things. vi, 5, 27, 36–40, 50–52, 60, 75

JSON JavaScript Object Notation. vi, 36, 37, 51, 52

LoRa Long Range. 4, 5, 10
LoRaWAN Long Range Wide-Area Network. 6
LPWAN Low Power Wide Area Network. vii, 4–6, 10, 60, 62

M-Bus Meter-Bus. 12, 17, 19
MCU Microcontroller Unit. 19, 20, 33, 35
MQTT Message Queuing Telemetry Transport. 6, 27, 36–38, 50

NB-IoT Narrow-Band IoT. 4–6, 10
NPW Negative Pressure Wave. 2, 8, 9, 11, 24
NWL Network Layer. 16, 19

OMS Open metering system. 2, 12, 16, 19, 62
OSI Open Systems Interconnection model. 12

PHY Physical Layer. 12

REST REpresentational State Transfer. 6, 40, 42
RF Radio Frequency. 19, 30, 32
RSSI Received Signal Strength Indicator. vi, 49, 56, 59

SDK Software Development Kit. 40, 42
SQL Structured Query Language. 38, 40, 47, 48, 51

TPL Transport Layer. 12, 15–18, 20, 29, 30

UART Universal Asynchronous Receiver-Transmitter. 32
USB Universal Serial Bus. 27

VIB Value Information Block. 18
VIF Value Information Field. 18
VIFE Value Information Field Extension. 18

WDN Water Distribution Network. 1, 2, 8–11, 20–25, 43, 45, 47, 50, 63
wM-Bus Wireless Meter-Bus. ii, iii, vi, vii, 2, 4–6, 8, 10–13, 19, 20, 26, 27, 29–33, 36, 49–51, 58–60, 62

Chapter 1

Introduction

1.1 Background and Motivation

Several municipals in Norway are starting to implement smart water meters to read water consumption in households, and this is something that Trondheim municipality also wishes to do. A smart meter is defined to be a meter which records information near real-time and transmits it wirelessly. This makes it possible to monitor water usage, reduce billing expenses and give more precise estimates compared to a more traditional approach of reading the values through manual inspection. By having the smart meters available in every household, it is possible to create an infrastructure to gather all data in a common platform. This could be more time and cost-effective.

As pipelines carrying water age, leakages inevitably occur. This results in increased costs in energy and chemical expenditure. The presence of leak and a quantification of loss can be measured straightforwardly at isolated parts of a Water Distribution Network (WDN), called District Metered Area (DMA). This is because a DMA only has one inlet pipe, and this one is installed with a flow meter, making it possible to detect leaks through a simple mass/volume balance analysis.

A more challenging task is to locate the leak within the DMA, which is sometimes a complex network of pipes. Most ways of localizing leakages has traditionally involved using some kind of hardware-based method to inspect only a smaller part of the DMA. This can end up being both costly and time-consuming [2]. By taking advantage of the emerging number of smart meters, there might be a way to implement a more cost-effective and less time-consuming way of localizing leakages. By also integrating pressure meters in combination with water consumption meters that's already starting to be implemented by Norwegian municipals, more knowledge of the WDN can be gathered and the number of possible leakage detection and localization strategies is increased.

This project seeks to solve these two challenges by suggesting an infrastructure able to collect near real-time water consumption and pressure measurements from several households in a neighborhood. This data will then be used to create a representational model of the DMA which generates pressure values that is compared with real pressure measurements to create pressure residuals. These pressure residuals are a vital step in further localization of leakages.

As Wireless Meter-Bus (wM-Bus) has been shown to be a popular European standard for smart meters, this will be the standard of choice for this project.

Previous efforts have been made by the same department to accomplish similar tasks. Lier H. [3] demonstrated an infrastructure for collection of wM-Bus data, but were only able to receive simulated data as communication with real meters could not be established. Vatland A. [4] made further efforts to establish communication with real meters, but were unsuccessful. A specialization project by Skjæveland C. G. [1] were able to establish communication with real meters through the use of new equipment by Radiocrafts as collector/receiver of wM-Bus data. The project serves as a continuation from these works.

1.2 Limitations

This report assumes that a Water Distribution Network (WDN) is divided into sub-regions called District Metered Area (DMA). Each DMA is isolated from the rest of the WDN with only a inlet where flow measurements are known. When flow measurements are available, leaks can be detected more easily since it is possible to establish simple mass balance in the pipes [5]. This paper is concerned with the localization of leaks within a DMA when a leak has been verified for the whole DMA. Here, it is assumed that both water consumption data and pressure data at each node is available, but flow measurements between the nodes in the pipes are not.

The meters available to measure flow and pressure at a node sends data every 95 second. This limits the applicability of leak detection methods that rely on fast transients, like the Negative Pressure Wave (NPW) method, which is described in chapter 2.

Moreover, Leak detection and localization methods described requires a network of several pressure and flow meters to be able to work. This project has only one real pressure meter and one real flow meter available. This project serves therefore as a demonstration of concept for infrastructure that is required and not as a final solution.

This project was conducted during the Covid-19 pandemic. This caused a limited availability of equipment placed out in the field. This includes wM-Bus collectors and wM-Bus meters. Fortunately, Professor Geir Mathisen has been of great assistance when physical intervention of the system has been needed.

The data available for the water grid has been limited. This has caused some inaccuracies in the implementation of the representational model of the water grid. Therefore, several assumptions have been made where it has been required. The demonstration of generating pressure residuals was shown, important for further leakage analysis.

1.3 Disposition of the Task

Chapter 2: Literary Study explores which communication technologies are viable for WDNs. Other smart water grid projects are investigated and a review of promising leakage detection strategies is performed.

Chapter 3: Theory describes the wM-Bus protocol, the OMS standard and how the STACK-FORCE protocol stack, used by SLWSTK6220A, is implemented. The theory behind water modelling done by the software EPANET is explained, properties of a leak is described and how

pressure residuals are used is described.

Chapter 4: Design and Specification sets up requirements for the system proposed. An overview of the whole system is also presented.

Chapter 5: Implementing the Infrastructure covers the implementation of all aspects of the system, such as the whole infrastructure and how data analysis is performed.

Chapter 6: Testing and Results explains the way the system was tested and what the results were.

Chapter 7: Discussion analyzes the results of the testing.

Chapter 8: Conclusion summarizes the main takeaways from the project.

Chapter 9: Further Work explores the ways in which this project can be built upon.

Chapter 2

Literary Study

A literary study is done to find the state-of-the-art concerning infrastructure solutions for smart water grids, communication technologies used for smart water grids, and also viable leakage detection and localization strategies. The findings are then summarized in the end.

2.1 Communication Technologies for Water Distribution Networks (WDNs)

A review article by Lalle Y. et al. (2021) [6] offers an overview of prominent communication technologies for smart water grids, as shown in figure 2.1. Here, Smart Water Grids are defined as water infrastructure integrating information and communication technologies. The article states that a lot of the new emerging technologies are of the type Low Power Wide Area Network (LPWAN). These promise to provide long range, low power consumption and good penetration capabilities for a low cost.

A. Pitì et al. (2017) [7] investigated suitable communication technologies for the roll-out phase of the Italian second generation of Smart Meters. Several use cases with various requirements were put forward. LPWAN technologies was seen as suitable for architectures using a public wide area network. Cellular networks, such as NB-IoT was argued to deliver higher service availablility because of using licensed bandwidth, but required involvement from a third-party. With a lot of users, these technologies would struggle to deliver data with a maximum sampling rate lower than 30 seconds and a maximum latency lower than 5 seconds. Maximum sampling rate of one minute and maximum latency of 10 seconds would be challenging, but feasible.

Even if there are many emerging LPWAN technologies, the main ones used as of 2020 are wM-Bus, Sigfox, LoRaWAN and NB-IoT [8]. These would probably be most readily available from manufacturers. This project uses Kamstrup as a supplier, and they do indeed provide these technologies for water meters along with linkIQ, which is their own specification based on wM-Bus. A comparison of these technologies are shown in table 2.1. Anani W. et al. (2019) [9] states that a choice depends on its use case. LoRa offers regional coverage with low duty cycle and low data rate for long battery life. it is suitable for private network without relying on a provider and supports mobility applications. Sigfox offers regional coverage with very low duty cycle and very low data rate for very long battery life. NB-IoT offers worlwide connectivity with very good coverage and medium data rate. It consumes more power due to the long

IoT Standards	LoRa/LoRaWAN	Sigfox	NB-IoT	WM-BUS
Frequency	Unlicensed Sub-GHz ISM	Sub-GHz ISM	Licensed 700-900 MHz	Unlicensed 868 MHZ, 433 MHz or 169 MHz
Data rate	0.3-37.5 kbps	0.1 kbps(UL), 0.6 kbps(DL)	150 kbps (NB) <1 mbps	2.4/4.8/19.2 kbps
Range for line of sight	5-15/20 km	30-50 km	22 km	500m (868MHz) and 5 km (169 MHz)
Coverage	157 dB	149 dB	164 dB	123 dB
Bandwidth	125 kHz - 250 kHz	100 Hz	180 kHz	335.5 kHz
Capacity	40,000	50,000	200,000	Not specified
Throughput	290 bps - 50 kbps	100 bps	250 kbps	4.8 - 100 kbps
Power	2µA resting, 12mA listening	0.01mA resting, Tx: 28mA, Rx: 10.5mA	3µA resting, Tx: 74-220mA, Rx: 46mA	0.6µA resting, Tx: 403mA, Rx: 31mA
Initiation	Node & Server	Node	Node	Node
Scalability	Medium	Low	High	Medium
Network Topology	Star	Star	Star	Star
Dedicated Network	No	Yes	Yes	No
Mobility/ Localization	Yes	Limited Mobility, No Localization	Limited mobility, No localization	No
Messages	unlimited	140 Message/day, 12 bytes/message	Unlimited	Not specified
Latency	1-2s (high downstream latency)	1-2s medium	1.5-10s	-
Modulation	CSS / GFSK	UNB/GFSK/BPSK	QPSK/OFDMA	FSK/GFSK/MSK/OOK/ASK
Bidirectional	Yes / Half-duplex	Limited / Half-duplex	Yes / Half-duplex	Yes
Standardization	LoRa Alliance	Sigfox Co.	3GPP	M-Bus

Table 2.1: A comparison of the currently most used LPWAN technologies for water infrastructure applications. Based on [9].

transmission. wM-Bus offers a large number of connectable devices, possibility for network expansion, fail-safe characteristics/robustness, minimum power consumption, and acceptable transmission speed. Suitable for remote meter reading, meter maintenance and configuration.

2.2 Smart Water Grid Projects

Lalle Y. et al. (2021) [6] performed a review of recent smart water grid projects in the period from 2007 to 2017. It was identified that the communication systems were based on conventional cellular networks, ZigBee and Bluetooth technologies for data communication. Because of energy and range considerations, it was recommended to use LPWAN technologies instead. NB-IoT was recommended for urban areas because of the large availability of cellular infrastructure, while other LPWAN technologies would require a gateway to interface to a server backend through cellular connection.

A unified framework for urban water management was proposed by G. Antzoulatos et al. (2020) [8], which exploited state-of-the-art IoT solutions for remote telemetry and control of water consumption in combination with machine learning-based processes. The solution presents a five-layered generic architecture, as shown in figure 2.2, that contextualizes the five goals of the architecture. The five goals are Smart Automated Metering & Remote Control, Device Connectivity & Data Management, Data Processing & Visualisation, Water Management, and Feedback of Water Usage.

The gateway, constituting the Device Connectivity Layer, is an important part of this system as it handles the connection between end devices and a central server infrastructure. for this project, a custom gateway was made that constitutes the backbone of a fixed wireless and multi-protocol network that handles meter readings and makes remote control viable. This gateway works for both wM-Bus and LoRa. This custom-made gateway came from the lack of commercially available gateways that suited their specific needs.

Communication with a central server is handled using the wireless internet protocols GSM

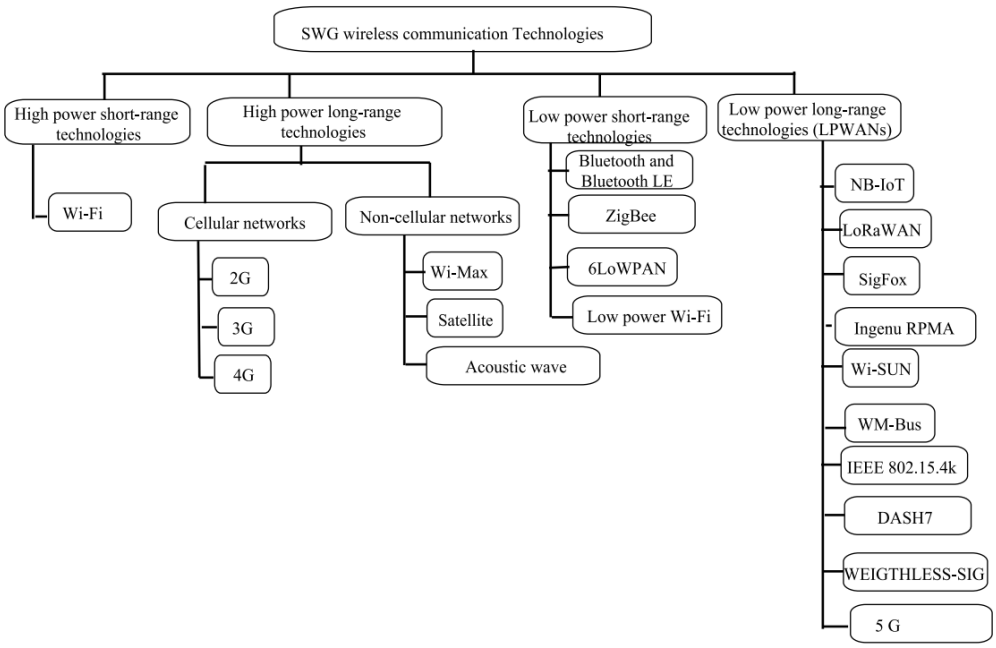


Figure 2.1: A classification of Smart Water Grid communication technologies. LPWANs are the ones of interest to this project. From [6].

and NB-IoT. The messages received by the central server uses the lightweight MQTT protocol. The packets are not decrypted/decoded until they reach the central server which is a part of the Device & Data Management Layer. The central server is based on the cloud service provided by Amazon Web Services (AWS). Here, a noSQL database is used for both storage and end device management. The servers Application Programming Interface (API) is based on a REpresentational State Transfer (REST) design with token-based authentication. This makes it possible for consumer's and administrator's applications to access the necessary data for their purposes.

The paper concludes that the gateways need to be positioned close to the end devices when wM-Bus is used. In this regard, LoRaWAN is far superior. The processing time upon gateway reception was less than a second on average, when serving up to 200 users in different scenarios.

2.3 A Review of Leakage Detection Strategies

Zaman D. et al. (2019) [2] provides a comprehensive review of different leakage detection strategies for pressurised pipes in steady state. In addition, they provide a way to categorise various leakage detection techniques, as shown in figure 2.3. It must be noted that in this case, leakage detection also involves leakage localization methods as well.

2.3.1 Classification of Leak Detection Strategies

Almost all leak detection techniques may be broadly classified as direct methods, indirect methods, and inferential methods. Most hardware-based approaches can be classified as direct methods of leak detection, whereas software-based methods are usually indirect or inferen-

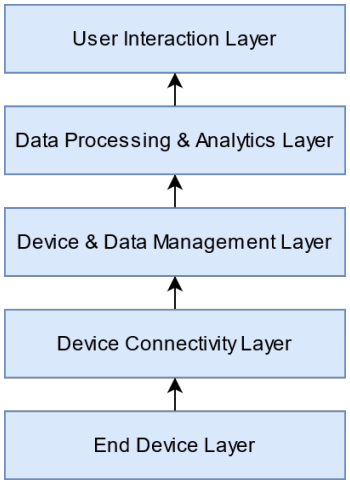


Figure 2.2: Five layered architecture for making a smart water management infrastructure. As suggested by [8]

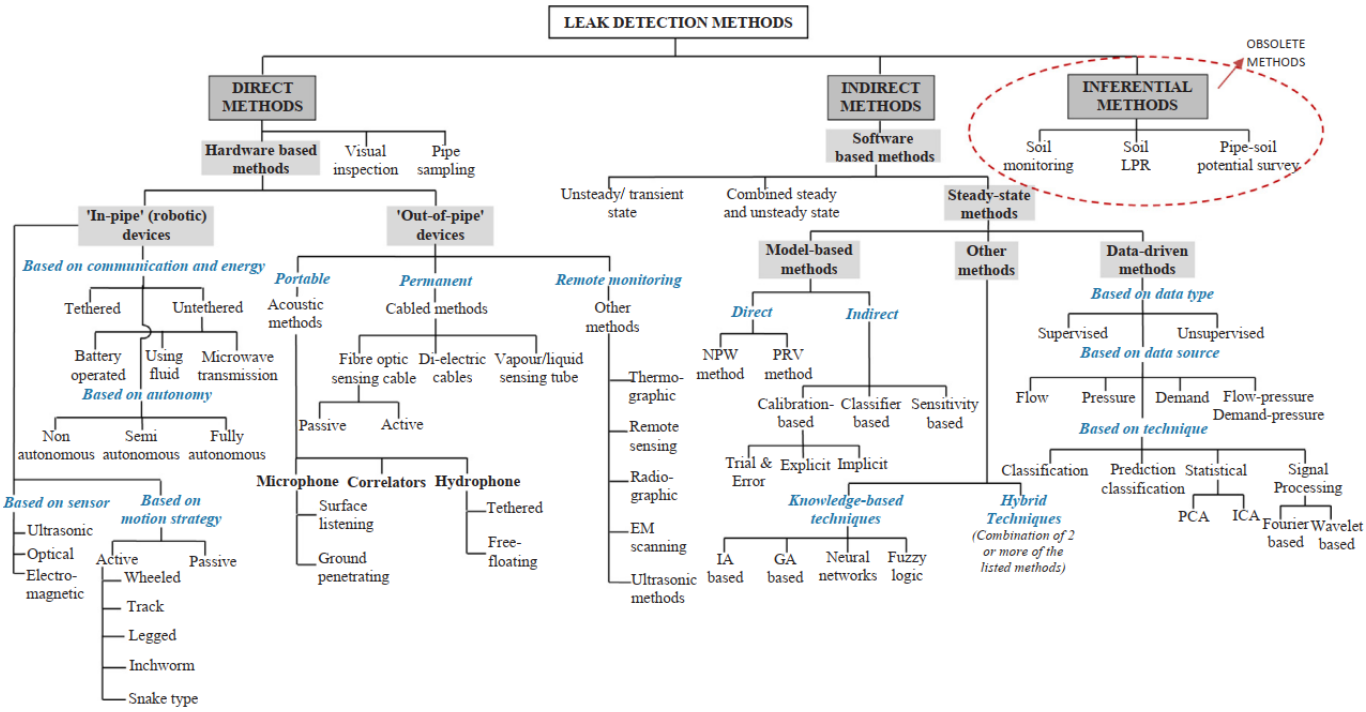


Figure 2.3: Classification of major leak detection techniques. From [2]

tial [2]. Inferential methods merely indicates the probable presence of a leak and doesn't aid leakage detection. They are rarely used nowadays and can be considered obsolete. Hardware-based approaches include several commercial 'in-pipe' (intrusive or robotic) and 'out-of-pipe' (external or non-intrusive) devices. These see a wide use today, unlike the software based, which are of interest to this project.

Software-based leakage detection techniques uses some form of mathematical technique performed by a software programming package. Most of these techniques can be considered either indirect or inferential. The hydraulic state of the pipe is important for a leak detection system. A technique may use either a transient-based approach, steady-state approach or a hybrid approach.

Steady-state approaches can be divided into two broad categories as data-driven or model-based. Model-based may be direct by using a simple and straightforward comparison of pressure, or indirect using complex mathematical operations to distinguish anomaly from pressure data. Data-driven techniques can be further divided based on type of data used, data source used and mathematical and computational technique used to extract leak information.

Software-based leak monitoring and detection approaches are still not popularly used by utility services as they are hugely data-intensive and require an adequate number of smart meters to be installed in the pipeline system. Hardware-based methods are instead more widely used [2].

Based on the classification provided by the paper, the system of this project would be most suited to a indirect method which is software-based. Presently, almost all software-based leak detection approaches utilised in real systems are operated under steady-state conditions [2]. Therefore, only steady-state approaches will be considered.

2.3.2 Mass/Volume Balance

When the flow data into a system is known, it is possible to apply conservation of mass principles. By assuming that the WDN is a closed system, it is possible to compare inlet and outlet flows. In a leak-free scenario, this should equal zero. A difference would then suggest that a leak or a undocumented outlet is present. If the WDN is comprised of isolated areas called DMA, then this method can be applied for only this part of the WDN. This way, a leak can be isolated to a specific DMA contained inside a WDN.

2.3.3 Negative Pressure Wave (NPW) Method

P. Ferrari et al. (2013) [10] considers the use of the Negative Pressure Wave (NPW) method to detect leaks in a wM-Bus meter network. the system measures the distance between two meters and a leak using the NPW that is generated. The time delay between the reception of the same NPW for the two meters can then be estimated. The distance between a meter 1 and the leak can be estimated using the equation:

$$L_1 = \frac{L + v_w \Delta t}{2} \tag{2.1}$$

where L is the distance between the two meters, v_w is the speed of the wave, and Δt is the time delay between the detection of the NPW between the two meters.

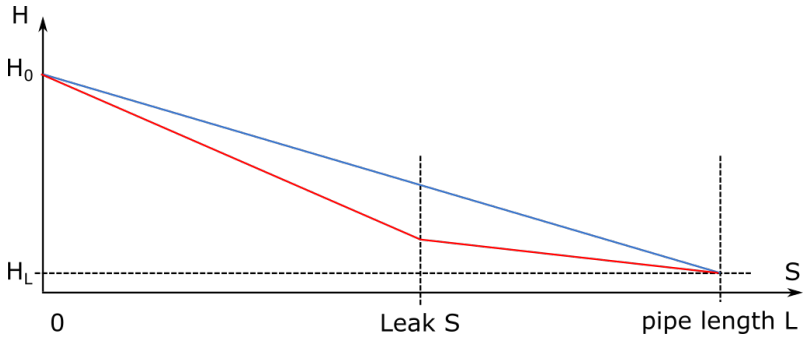


Figure 2.4: A graph of pressure drop along a pipe section. The same pressure drop occurs with and without leak, but the line changes. Red line is the pressure drop when a leak is present, blue line is the pressure drop without a leak present. x-axis is length and y-axis is pressure head. Based on [11]

The speed v_w of a NPW depends on many factors, but can easily reach 1000 m/s. A time synchronization uncertainty below 1 ms would then be required to obtain a position uncertainty on the order of 1 m [10].

2.3.4 Gradient Intersection Method

Fiedler J. (2014) [11] covers the gradient intersect method. The method assumes a linear pressure drop along a pipe section, as shown in figure 2.4. When a leak occurs, the flow increases before the leak while decreases after. This results in a increased pressure drop before and reduced pressure drop after. These two pressure lines can then be used to find an intersection which gives the location of a leak.

2.3.5 Using Pressure Residuals From a Model-Based Approach

Casillas M. V. et al. (2013) [12] describes the use of pressure residuals for leakage diagnosis, which is the underlying philosophy for all model-based methods. Pressure residuals are the difference in simulated and real pressure values at the same location for a simulated and real network. Such a method requires extensive knowledge about the WDN as well as calibration and verification of the model for it to work effectively. The residuals can be used for several leakage detection and leakage localization methods. Casillas M. V et al. (2013) [12] compares five model-based leakage localization methods and concludes that the angle method increases the capability of isolating leaks in a great number of cases. Noise was a great source of error. Other works have noted that model errors and closed valves also obscures leak localization [12].

2.3.6 Mixed Model-Based/Data-Driven Approach

D. Zaman et al. (2019) [2] concluded in its review of leakage detection methods that a hybrid method provides a better performance concerning the accuracy in detection and lesser error in terms of false alarm. Soldevila A. et al. (2016) [5] proposes such a method. The problem of using raw pressure residuals is that leaks affect all residuals to some extent in addition to having a high uncertainty. A classifier is therefore suggested to solve this problem. The presence of leakage in determined for a DMA by using mass/volume balance. Then, localization of the leakage is done by obtaining pressure residuals for the DMA. These are finally used by a classifier, similar to a data-driven approach. Soldevila A. et al. (2016) [5] argues that a

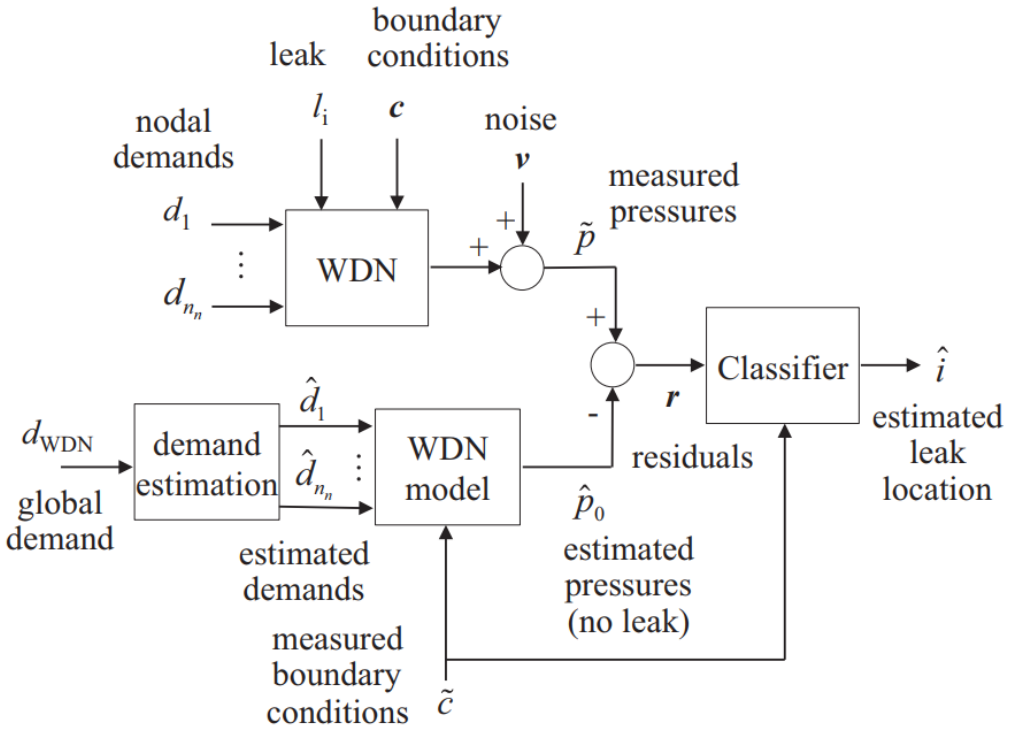


Figure 2.5: Leak localization scheme based on a hybrid approach. Pressure residuals are generated from a comparison between a model and the WDN. These are used by a classifier to state possible leak locations. From [5]

time horizon of 24 hours is sufficient for maximum localization accuracy of 90 percent. The size of the leaks were 50 litres per second. A sampling rate of 10 minutes was used, were six samples were averaged to create hourly samples with less uncertainty. K-NN nearest neighbour was used as a classifier. Although, the paper suggests that more effective classifiers should be investigated. This scheme is laid out in figure 2.5. This scheme will be further detailed in chapter 3.6, as it is similar for only generating pressure residuals.

2.4 Summary

Recent papers concerning communication technologies for smart water grids calls for the use of LPWAN to provide a solution with low energy usage and long range. There are some concerns raised for the use of wM-Bus since it has the lowest range of the popular LPWAN technologies [8]. Both NB-IoT and LoRa gets a lot of recommendations to be used in future smart water grids. What wM-Bus can provide is a higher data rate with a lower power consumption. This can be beneficial for certain leakage detection and localization strategies. The generic architecture model, proposed by G. Antzoulatos et al. [8] and shown in figure 2.2, provides a clear view of a water management system. It will therefore be used for this project, as shown with a system overview in chapter 4.

The literature provides a vast number of leakage detection and localization strategies, but the best choice depends on the properties of the system. A former study of the Trondheim water grid notes that the network is divided into 55 DMA, each one with a flow meter that measures how much water enter by the inlet pipe [13]. For this reason, leakage detection is possible by

using mass/volume balance for each DMA.

Inside the DMA, there are no flow measurements between each node, only consumption and pressure data at each node. The choice is limited to software-based methods as it is desirable to have a leakage localization method that is non-intrusive and can be active at all times. In the literature, the most popular software-based ones are steady-state [2]. There is a choice between model-driven, data-driven or hybrid methods. Data-driven techniques can be suitable for water grids with an ample amount of meters, and it wouldn't require extensive knowledge about the water grid. On the other hand, it would only be able to deal with faults that have been previously experienced by the system [5]. A model-based method would require extensive knowledge about the system, but it would be possible to generate a lot of different leakage scenarios to compare to real meter data. A hybrid method increases the complexity of the leakage detection algorithm, but has shown to provide better performance.

The use of wM-Bus limits pressure wave analysis methods, such as the NPW method, since this requires a very high sampling rate and very low time synchronization uncertainty. The complex layout and size of WDNs increases the frictional losses, and this non-linearity makes some detection strategies less viable. Examples are NPW method and gradient intersection, which are more suitable for straight pipelines. From these considerations, this project will therefore address a model-driven approach where a model of the DMA is created. The data generated from this model will then be used for generating pressure residuals by comparing it to real measurements. This is further described in chapter 3.6. Similarly to D. Zaman et al. (2019) [2], about 6 samples will be averaged to create samples every 10 minutes.

Chapter 3

Theory

This chapter will present the theory needed in order to understand how the system in this project works. The topics covered is the Wireless Meter-Bus (wM-Bus) protocol which is used by all metering equipment, the Open metering system (OMS), which is the more specific subset of wM-Bus used by the devices. The STACKFORCE protocol stack will be covered, which is the wM-Bus implementation used by the SLWSTK6220A starter kit for simulated meters. The theory behind the water model used by EPANET is covered, the hydraulic properties of a leak is described and lastly, the generation of pressure residuals is explained.

3.1 The Wireless M-Bus Standard

The Meter-Bus (M-Bus) protocol is a European standard for remote reading of various consumption meters, as well as sensors and actuators. It has a wireless implementation called Wireless Meter-Bus (wM-Bus), which replaces the Physical Layer (PHY) and Data Link Layer (DLL) of the M-Bus layer model. This is the version used in this project. A wM-Bus transceiver offers low power usage because of its small overhead while also offering unidirectional modes, which reduces power usage further by not having to worry about data being sent both ways. It also has a long range because of frequencies being under 1 GHz. The M-Bus protocol is described in the standard EN 13757.

The wM-Bus protocol is compliant with the Open Systems Interconnection model (OSI). Table 3.1 shows the relationship between the OSI model and the wM-Bus protocol. Physical Layer (PHY) and Data Link Layer (DLL) is always present. Transport Layer (TPL) and the applied Application Layer (APL) are always introduced by the TPL's Control Information (CI) field. Optional layers like Extended Link Layer (ELL) and Authentication and Fragmentation Layer (AFL) are introduced by special CI-fields. In such cases the M-Bus-message contains several CI-fields [14].

A overview of all wM-Bus packets for this project is shown in appendix A.

3.1.1 Physical Layer (PHY)

The protocol uses a half-duplex asynchronous serial transmission with a Master-Slave structure. Meters transmit metering data by RF in regular intervals that is received by a Master/-

Abbrevation	Layer	Described in	OSI model Layer
APL	Application Layer	EN 13757-3	Application Presentation
TPL	Transport Layer	EN 13757-7 / EN 13757-3	Session Transport
AFL	Authentication and Fragmentation Sublayer	EN 13757-7	
NWL	Network Layer	EN 13757-5	Network
ELL	Extended Link Layer	EN 13757-4	Data Link
DLL	Data Link Layer	EN 13757-4	
PHY	Physical Layer	EN 13757-4	Physical

Table 3.1: The wM-Bus protocol as a layer model. As shown, the relationship between M-Bus layers and the OSI layers is not one-to-one. Based on table 1 from [15]

Mode	Frequency [MHz]	Description of Use
S1	868.3 / 433	Send data a few times per day. Optimized for battery operation and stationary operation
S2	868.3 / 433	Same as S1, but bidirectional communication
T1	868.95 / 433	Send data every few seconds. Configurable interval
T2	868.95 / 868.3 / 433	Same as T1, but bi-directional
C1	868.95 / 433	Uses NRZ coding. Similar to T1 but higher data-rate
C2	869.525 / 868.95 / 433	Same as C1, but bidirectional
N1a-f	169	Narrowband communication for long range transmission
N2a-f	169	Same as N1a-f, but bidirectional
N1g	169	Narrowband communication for long range transmission
N2g	169	Same as N1g, but bidirectional

Table 3.2: This table lists the most common operating modes for wM-Bus

Collector. The collector may also query data from bidirectional wireless M-Bus meters.

The wM-Bus operates under various modes, as shown in table 3.2. Only the most common modes have been listed. The modes define the configuration of the radio channel and also the communication flow.The various modes vary in data encoding, frequency, data rate and frequency modulation. A mode can support either bidirectional or unidirectional communication. In a unidirectional mode, less overhead is required and can give less power consumption. A meter (slave) will then only be able to send data packets to a collector (master). In a bidirectional mode, a collector is able to request data when it is required.

For mode C1, which is used by this project, all communication is NRZ-encoded. All communication from a collector to a meter is transmitted as FSK modulated data. All communication from a meter to a collector is transmitted as GFSK modulated data. All communication is preceded by a preamble and synchronization pattern which helps the decoder determine which mode is used.

First block: Data Link Layer						
Length	Ctrl	Manuf	Address	Version	Type	CRC
1 byte	1 byte	2 bytes	4 bytes	1 byte	1 byte	2 bytes

Second block: Application layer		
CI	Data	CRC
1 byte	up to 15 bytes	2 bytes

Optional block: Application layer	
Data	CRC
up to 16 bytes	2 bytes

Table 3.3: M-Bus frame format A.

First block: Data Link Layer					
Length	Ctrl	Manuf	Address	Version	Type
1 byte	1 byte	2 bytes	4 bytes	1 byte	1 byte

Second block: Application layer		
CI	Data	CRC
1 byte	up to 115 bytes	2 bytes

Optional block: Application layer	
Data	CRC
up to 126 bytes	2 bytes

Table 3.4: M-Bus frame format B.

3.1.2 Data Link Layer (DLL)

The Data Link Layer follows immediately after the preamble and synchronization pattern. Two different frame formats are supported, called frame format A and frame format B. Frame format is determined by preamble and synchronization pattern. An overview of both frame formats are shown in table 3.3 and table 3.4 respectively. At the very least, the frame formats consists of two blocks, one defined by DLL and one by APL. Frame format B does not have a CRC-field for the first block while also being able to contain more data. Frame format B is used for this project. A description of all DLL fields follows:

Length (L) Field

The L-field specifies the number of bytes contained in the data packet. For frame format A, CRC bytes should be excluded, but not for frame format B.

Control (C) Field

The C-field is used to declare the message type. The 8 bit field is shown in table 3.5. RES is always 0. A value of 1 for PRM means that message comes from a initiating station, while 0 is a responding station. FCB, FCV and ACD, DFC-bit coding is described in EN 60870-5-2.

MSBit				LSBit	
RES	PRM	FCB	FCV	Function code	Primary to secondary
		ACD	DFC		Secondary to primary

Table 3.5: C-field data format. From [16]

M1	**M2**	
2D	2C	Kamstrup
24	48	Radiocrafts
9A	CE	Silicon Labs

Table 3.6: Manufacturer ID's used in this project.

Several message types exist. For this project, the only message type used is 0x44, meaning "Send spontaneous/periodical application data without request (Send/No Reply) from an initiating station". This is mandatory for modes like S1, T1, C1 and N1.

Manufacturer (M) Field

The M-field is a unique two-byte ID to identify the supplier of the equipment sending the data packet. The ID's used in this project is listed in table 3.6.

Address (A) Field

The A-field contains the address of the sender. A manufacturer needs to ensure a uniqueness of the addresses for all produced meters [16].

Version (V) Field

V-field specifies the version of the meter. It makes sure that each version of a device has its own unique ID.

Type (T) Field

T-field gives information about what type of meter that sent the package. The standard EN 13757-7:2018 [15] specifies various device type codes. The ones used in this project are listed in table 3.7.

Control Information (CI) Field

The CI-field specifies the structure of the next higher protocol layer. The block declared may be an APL, AFL, TPL or ELL. If ELL or AFL is declared, then several blocks may be chained,

Type	
0x07	Water Meter
0x16	Cold water meter
0x18	Pressure Meter

Table 3.7: Device types used in this project.

CI-field	Layer	TPL Header	Used by Meter	Description
0x78	APL	None	Kamstrup	Send full M-Bus frame
0x79	APL	None	Kamstrup	Send compact M-Bus frame
0x7A	APL	Short	SLWSTK6220A	Send full M-Bus frame
0x8D	ELL	-	Kamstrup	Add additional link layer of 8 bytes

Table 3.8: CI-fields used for this project. ELL can't use a TPL header and is therefore unspecified.

and the data packet will contain several CI-fields. The values used for this project are shown in table 3.8.

Cyclic Redundancy Check (CRC) Field

The CRC-field is used to perform a cyclic redundancy check to detect errors in the generated data packets as they are received. It is computed over the information from the previous block. The CRC formula used is:

$$x^{16} + x^{13} + x^{12} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^2 + 1 \tag{3.1}$$

An inital value of 0 is used while the final CRC is complemented.

3.1.3 Extended Link Layer (ELL)

The Extended Link Layer (ELL) is an extension of the DLL and provides additional control fields for wireless communication. Several ELL blocks are defined in the standard EN13737-4 [16]. For this project, the Kamstrup meters use an 8 byte block with CI value of `0x8D`, which contains the following information:

8 Byte Extended Link Layer			
CC	ACC	SN	CRC
1 byte	1 byte	4 bytes	2 bytes

The **Communication Control (CC)** field provides information about how a unit should respond to a message, if the message is syncronized, if the message should be prioritized and if it is a repeated message. The **Access Number (ACC)** field is used to synchronize transmission from the meter. The **Session Number (SN)** field contains the following information:

Session Number (SN)		
ENC	T	S
3 bits	25 bits	4 bits

The **Encryption (ENC)** subfield is three bits long and defines which encryption mode is used. As of 2021, only AES-128 Counter Mode is available, which is defined by `0b001`. The **Time (T)** subfield is 25 bits and describes a relative time for when the packet was sent in number of minutes. with 25 bits, it is able to count to almost 64 years. The **Session (S)** subfield is 4 bits and defines the session number within a minute of the T-subfield. With 4 bits, it is able to have 16 sessions within a minute.

3.1.4 Network Layer (NWL)

An additional Network Layer (NWL) can be defined between ELL and AFL, and is described in EN 13757-5. This is never used by OMS [14] and not implemented by any of the equipment

in this project.

3.1.5 Authentication and Fragmentation Layer (AFL)

The standard considers Authentication and Fragmentation Layer (AFL) to be a part of TPL. Its functions are to fragment longer messages into several packets, prove authenticity of TPL and APL with the use of a Message Authentication Code (MAC) and a message counter for identification and security features. This layer is added if one of these features are required. This is not the case for any of the equipment in this project.

3.1.6 Transport Layer (TPL)

For the transport of APL information, a TPL header structure can be chosen. The header structures are:

- No TPL-header: Used for unencrypted messages. Data follows right after CI-field:

TPL
CI
1 byte

- Short TPL-header: Contains access number, status byte and configuration field. A 2 byte sequence 0x2F 0x2F decryption verification (DV) follows:

Transport Layer (TPL) Header				
CI	ACC	STS	CF	(DV)
1 byte	1 byte	1 byte	2 bytes	2 bytes

- Long TPL-header: Contains access number, status byte and configuration field. It also contains meter identification. A 2 byte sequence 0x2F 0x2F decryption verification (DV) follows:

Transport Layer (TPL) Header								
CI	Addr	Manuf	Ver	Type	ACC	STS	CF	(DV)
1 byte	4 bytes	2 bytes	1 byte	1 byte	1 byte	1 byte	2 bytes	2 bytes

Table 3.8 shows which block types uses which TPL header format. A long header may be used if there is a requirement of supplying a different address than what is specified in the DLL from the manufacturer. If not, a short header is sufficient. The **Access Number (ACC)** field is incremented for each packet and may be used for synchronization purposes. The **Status (STS)** field gives information of errors that might arise in addition to manufacturer specific statuses. The **Configuration (CF)** field gives information about the security mode used and the length of the encrypted data.

3.1.7 Application Layer (APL)

M-Bus Frame

Data can be structured in different ways by the APL. These are as a **full M-bus frame**, **compact M-Bus frame** or as a **M-Bus format frame**. The full M-Bus frame transmits data information fields alongside the transmitted data. The compact M-Bus frame is able to reduce package size by 40 percent by omitting this. An M-Bus format frame transmits only the data information fields. The full and compact M-Bus frame is used in this project and is shown in figure 3.9.

Full M-Bus Frame								
CI	Data header	DIF[1]	VIF[1]	Data[1]	DIF[2]	VIF[2]	VIFE[2]	Data[2]

Compact M-Bus Frame					
CI	Data header	Format Dignature	Full-Frame-CRC	Data[1]	Data[2]

Table 3.9: Full and compact M-bus frame.

DIF	
05	Instantaneous value. 8 digit BCD (binary coded decimal)
02	Instantaneous value. 16 bit integer/binary
04	Instantaneous value. 32 bit integer/binary
44	Stored Instantaneous value. 32 bit integer/binary
61	Minimum value. 8 bit integer/binary
22	Minimum value. 16 bit integer/binary
12	Maximum value. 16 bit integer/binary

Table 3.10: The data information for data records in this project. Stored instantaneous value is the monthly flow which is updated at the start of every month. The values are in hex number system.

Encryption

The Wireless M-Bus standard allows for the encryption of the payload data using optimized encryption modes like AES 128 CBC. It can be defined either by the ELL or by the TPL header.

Data and Value Information Blocks

Each data record has a Data Information Block (DIB) and a Value Information Block (VIB) associated with it. The DIB describes the length, type and coding of the data, while the VIB specifies the unit and multiplier of the data. The DIB contains the Data Information Field (DIF) and the Data Information Field Extension (DIFE), while the VIB contains the Value Information Field (VIF) and the Value Information Field Extension (VIFE). The values used for this project are shown in table 3.10 and table 3.11.

VIF	**VIFE**	
13		Volume $10^{-5}\ m^3$
67		External temperature $10^{-1}\ C$
69		Pressure $10^{-1}\ bar$
FF	09	Manufacturer Specific
FF	0A	Manufacturer Specific
FD	17	Binary Error flags
FF	20	Manufacturer Specific

Table 3.11: The value information for data records in this project. The manufacturer specific values are unknown as no information can be found. The values are in hex number system.

3.2 The Open Metering System Standard

The Open metering system (OMS) standard is an effort in making a manufacturer- and utilities-independent standardization for M-Bus and ensure interoperability between all meter products. The standard implements a subset of the highly flexible M-Bus standard as specified in EN13757. Where the M-Bus standard leaves ambiguity, OMS offers a more precise definition. The OMS group consists of a number of member companies, of which Kamstrup, STACKFORCE and Radiocrafts are some of them. Some important requirements made by OMS are:

- The OMS standard requires the use of only S1, S2, T1, T2, C1 and C2, all operating on frequency 868 MHz to 870 MHz.
- The modes S1, S2, T1 and T2 is considered deprecated, and therefore not recommended.
- All wireless and Programmable Logic Controller (PLC) communication have to be encrypted.
- The use of frame format A, as shown in table 3.3, is required.
- DLL encryption shall not be applied.
- A packet shall not transmit several datagrams. Instead, AFL shall be used to fragment longer messages.
- Only a subset of C-fields are supported.
- The NWL shall not be used.
- Only a short ELL or a long ELL is supported.

Even though Kamstrup is a member of the OMS group, the wM-Bus packets from the Kamstrup meters deviate from the OMS requirements in two ways: Frame format B is used instead of frame format A, as evident by no CRC being transmitted, and an 8 byte ELL is used instead of the two required by OMS.

3.3 The STACKFORCE Protocol Stack

The Starter Kit by Silicon Labs, called SLWSTK6220A, uses a protocol stack called STACKFORCE. This protocol stack implements three main parts; a Hardware Abstraction layer (HAL), a RF driver and the wM-Bus stack [17]. An overview of the protocol stack is shown in figure 3.1. As the figure shows, the various layers are interfaced by using different APIs that are defined.

The HAL makes an abstraction for all hardware resources that is required by the protocol. The function `E_HAL_STATUS_t wmbus_hal_init(void);` declared in `wmbus_hal.h` initiates the initialization routines of all required Hardware Abstraction layer (HAL) modules, including the RF driver.

The RF driver implements the hardware abstraction for the radio driver. A choice of driver needs to be made based upon which MCU is used, which wM-Bus mode is used, and if the device is defined as a collector or meter. The driver is defined as a static library called `librf.a`.

The wM-Bus stack implements all layers of the protocol, as shown in table 3.1. It is compliant to both the wM-Bus protocol and also the OMS standard. In addition, there is a serial interface on top which gives access to every single function in the APL through a serial port connection. The protocol stack is available as a static library called `libstack.a`. There are several imple-

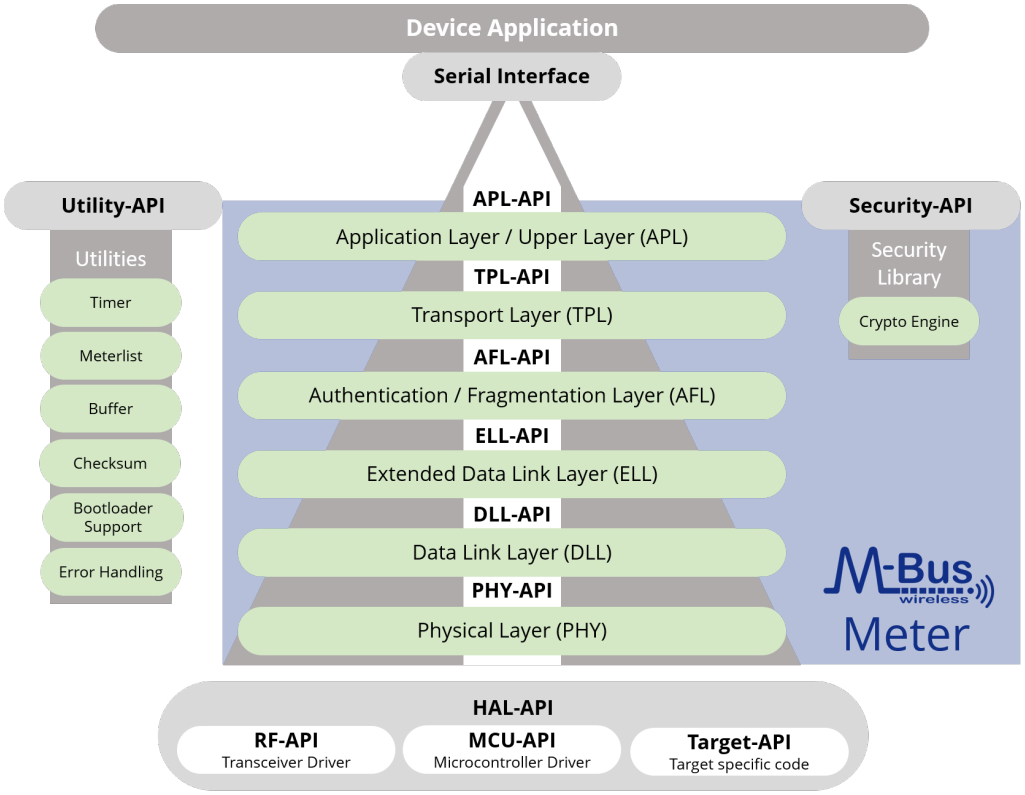


Figure 3.1: Architecture of the STACKFORCE wM-Bus protocol stack. From [18]

mentations of the protocol, based upon which MCU is used, which wM-Bus mode is used, and if the device is defined as a collector or meter. In addition, some have APL as the top layer, and some have TPL as the top layer. A stack with less layers implemented might be desired if a custom layer implementation is needed.

Several IAR workspaces are provided by STACKFORCE that includes all files needed for various use cases. A main c file is then used to initiate the wM-Bus stack and HAL.

3.4 EPANET and water modelling

EPANET is a widely used software application to model water distribution networks, both by engineers and researchers. The software was developed by United States Environmental Protection Agency's (EPA) and made free to download. The software uses mathematical concepts to develop its model which will be covered in this section.

3.4.1 The EPANET Model

An EPANET model is made for a District Metered Area (DMA), which is a small part of the Water Distribution Network (WDN), as shown in figure 3.2. A DMA is isolated from the rest of the grid using an isolation valve where a flow meter is installed. It is therefore not necessary to create a model of the whole WDN.

Each node is a place where there exists an outlet for water. With this model, it is not possible to have leaks in the pipes in between, and a leak needs to be localized in one or several nodes.

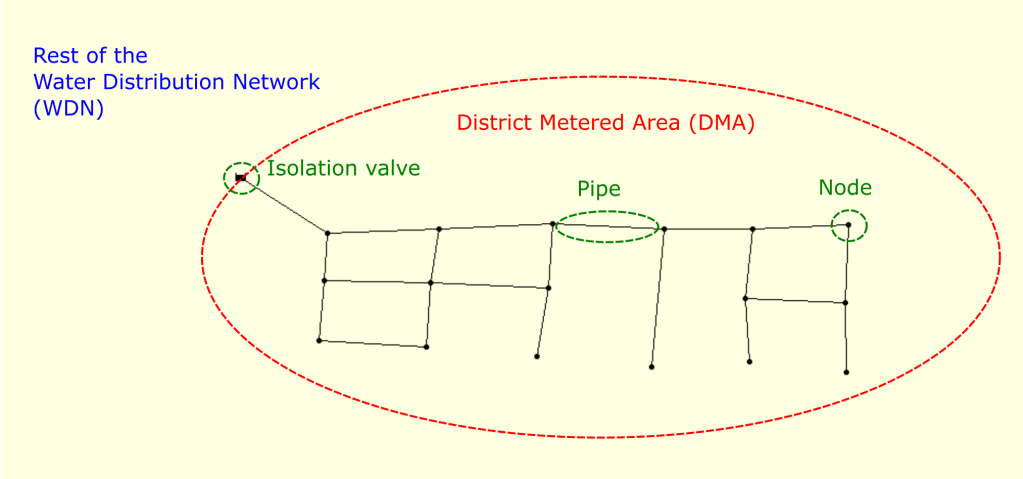


Figure 3.2: Overview of vocabulary concerning the model. The isolation valve separates the DMA from the rest of the WDN.

The properties tied to nodes are water consumption and total head H, and the property tied to pipes are flow Q. Head can be thought of as the energy that relates to an equivalent static liquid column.

3.4.2 Conservation of Energy and Mass

For fluids being transported in pipes, the conservation of energy in the system from point i to point j is taken advantage of in order to produce models of good accuracy. The relationship between potential energy, static pressure and kinetic energy is stated in Bernoulli's principle:

$$H = z + \frac{p}{\rho g} + \frac{v^2}{2g}, \tag{3.2}$$

where H is total head, z is height over reference value, p is pressure at a chosen point, ρ is density of a fluid, g is acceleration of gravity and v is fluid flow speed at a point on a streamline.

The velocity head is $\frac{v^2}{2g}$ and pressure head is $\frac{p}{\rho g}$. For Water Distribution Network (WDN), the contribution of velocity head is very small compared to pressure head and height. Therefore, EPANET omits velocity head in its calculations:

$$H = z + \frac{p}{\rho g} \tag{3.3}$$

The contribution of friction is substantial in WDN. The head loss along a pipe from node i to node j can be stated, using equation 3.3, as:

$$z_i + \frac{p_i}{\rho g} = z_j + \frac{p_j}{\rho g} + H_L(k), \tag{3.4}$$

where H_L is head loss due to friction and k = index of the pipe connecting node i and node j.

EPANET offers three ways to calculate head loss, where each is an equation based on an empirical relationship which relates the flow of water and pressure drop with the physical properties of the pipe. These are named Hazen-Williams equation, Darcy-Weisbach equation and

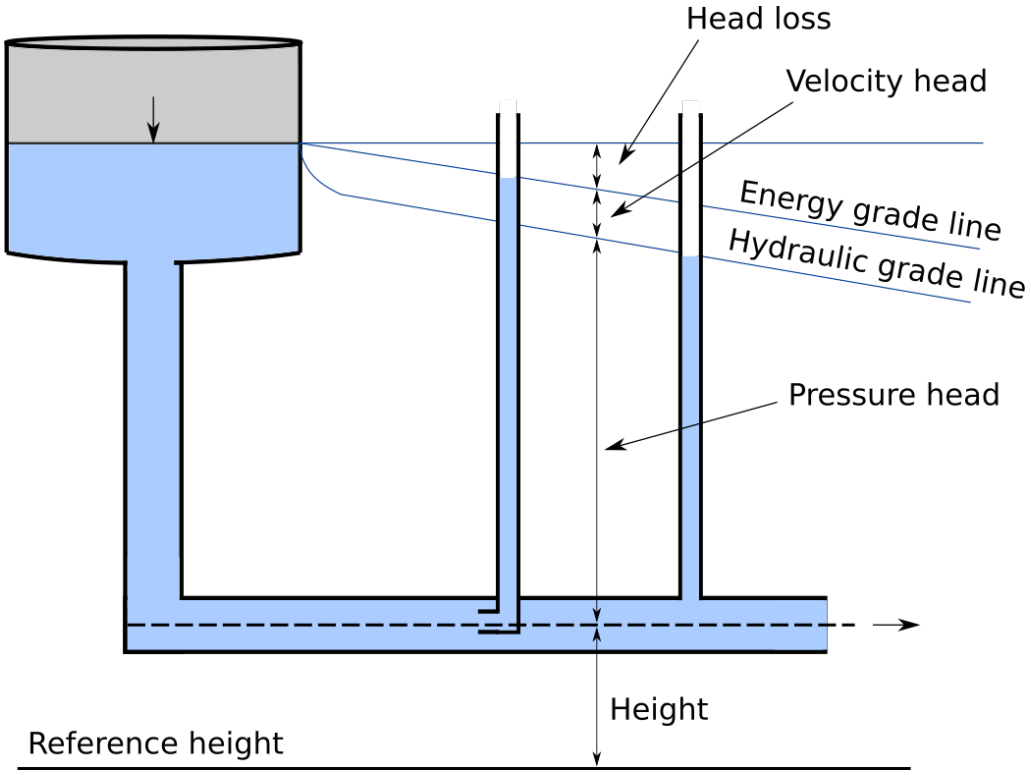


Figure 3.3: Total head along a pipeline with contribution from height, pressure head and velocity head. Head loss is the lost pressure due to friction. Velocity head is assumed to be zero. Inspired by [19]

the Chezy-Manning equation. In this project, the Hazen-Williams equation is used. It has the advantage of having a coefficient that is not the function of the Reynolds number. Although, it is only applicable for water at conventional velocities. The water is assumed to be at room temperature, which is a source of error when temperatures are different from this. The equation is:

$$H_L(k) = R_k Q_k^{1.852} = \frac{10.67 L_k}{C_k^{1.852} d_k^{4.8704}} Q_k^{1.852}, \tag{3.5}$$

where L is the length of a pipe section, C is the roughness coefficient, d is the diameter of the pipe and Q is the flow through the pipe segment. R is the static resistance factor. k = index of a pipe connecting two nodes.

By using equation 3.3, equation 3.4 and equation 3.5, the conservation of energy can be stated as:

$$-R_k Q_k^{0.852} Q_k + H_i - H_j = 0 \tag{3.6}$$

In addition to conservation of energy, the conservation of mass for a WDN is upheld by requiring that each total inflow equals the total outflow for every network node. This can be expressed as a linear set of equations:

$$\sum_{k=1}^{n_i} Q_{k_{i,j}} + q_i = 0 \tag{3.7}$$

where $Q_{k_{i,j}}$ = flow in the pipe $k_{i,j}$ from node i to j, n_i = number of pipes connected to node i, and q_i = known demand at node i.

3.4.3 Analysis Algorithm to Find Total Head and Flow

Both conservation of energy [3.6] and conservation of mass [3.7] is used to represent the WDN. The unknown variables are the flow through pipes Q and nodal heads H. To find these unknowns, a modified Newton-Raphson method is used, called Todini's Global Gradient Algorithm (GGA) [20].

First, Todini's modified Gradient Algorithm (GA) respresentation of WDN is used. This is simply a restatement of equation 3.6 and equation 3.7 in a matrix notation [21]:

$$\left[\begin{array}{c|c} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \hline \mathbf{A_{21}} & \mathbf{0} \end{array}\right]\left[\begin{array}{c} \mathbf{Q} \\ \hline \mathbf{H} \end{array}\right]+\left[\begin{array}{c} \mathbf{A_{10}H_0} \\ \hline \mathbf{q} \end{array}\right]=\left[\begin{array}{c} \mathbf{-F_1} \\ \hline \mathbf{-F_2} \end{array}\right] \tag{3.8}$$

$\mathbf{A_{11}}$ is a diagonal matrix whose elements are defined as:

$$\mathbf{A_{11}}(k,k) = A_k = R_k Q_k^{0.852} \tag{3.9}$$

$\mathbf{A_{12}}$ is a [p;n] matrix which relates the pipes to the unknown head nodes, and $\mathbf{A_{21}} = \mathbf{A_{12}^T}$. $\mathbf{Q}$ are all the unknown pipe flows and $\mathbf{H}$ are all the unknown nodal heads. $\mathbf{A_{10}}$ is a $[p; n_0]$ matrix relating the pipes to the fixed head nodes. $\mathbf{q}$ are the known nodal demands. $\mathbf{F_1} = \mathbf{F_1}(\mathbf{Q},\mathbf{H})$ and $\mathbf{F_2} = \mathbf{F_2}(\mathbf{Q})$ indicate how far from zero the relevant equations are for any guessed solution $\mathbf{Q}$ and $\mathbf{H}$. Applying the Newton Raphson method to equation 3.8 results in the following system of equations with flow and head correction for iteration τ:

$$\left[\begin{array}{c|c} \mathbf{D_{11}^\tau} & \mathbf{A_{12}} \\ \hline \mathbf{A_{21}} & \mathbf{0} \end{array}\right]\left[\begin{array}{c} \mathbf{dQ} \\ \hline \mathbf{dH} \end{array}\right]=\left[\begin{array}{c} \mathbf{-F_1^\tau} \\ \hline \mathbf{-F_2^\tau} \end{array}\right] \tag{3.10}$$

where

$$\mathbf{dQ} = \mathbf{Q}^\tau - \mathbf{Q}^{\tau+1} \tag{3.11}$$

$$\mathbf{dH} = \mathbf{H}^\tau - \mathbf{H}^{\tau+1} \tag{3.12}$$

$$\mathbf{F_1^\tau} = \mathbf{A_{11}^\tau Q^\tau} + \mathbf{A_{12}^\tau H^\tau} + \mathbf{A_{10}H_0} \tag{3.13}$$

$$\mathbf{F_2^\tau} = \mathbf{A_{21}Q^\tau} + \mathbf{q} \tag{3.14}$$

and $\mathbf{D}_{11}$ is a diagonal matrix which is the Jacobian of $\mathbf{A}_{11}\mathbf{Q}$. The diagonal elements are:

$$\mathbf{D_{11}}(k,k) = D_k = 1.852 R_k Q_k^{0.852} \tag{3.15}$$

A compact expression of equation 3.10 is:

$$\mathbf{Ah} = \mathbf{-F} \tag{3.16}$$

To find a solution, the system needs to be solved for $\mathbf{h} = \mathbf{A^{-1}}(\mathbf{-F})$. A guess for all pipeline flows $\mathbf{Q}$ and nodal head $\mathbf{H}$ are made, and $\mathbf{h}$ then gives an indication of how the guess needs to be modified. With enough iterations, an acceptable solution is found.

3.4.4 Modelling Water Demand Patterns

A Water demand pattern is a list of flow estimation Q for a certain time period defined for the model. They are usually obtained by using water consumption data of a DMA inlet, distributed for each node based on historical billing records [5]. A better method is to use actual measurements from each node using smart meters for water consumption.

A problem of using volume-based meter readings, is that the duration of the water consumption is not known when the sampling rate of the meter is lower. When the volume readings are averaged on time to get litres per second, the flow will be much lower than in reality. As pressure in the pipes is greatly affected by flow Q, there will be a smaller change in pressure for a model compared to pressure readings when tapping occurs. Calibration can be performed to match the mode. This involves adding a multiplier for the water consumption used by the model.

The Kamstrup Pressuresensor provides a minimum and maximum pressure reading for an interval in addition to the instant pressure reading at each interval. It is possible to use this extra information to make assessments of the tapping event. This could be used to make better estimations of the water demand pattern.

3.5 Hydraulic Properties of a Leak

By using smart meters, it is possible to gather information about hydraulic properties in a Water Distribution Network (WDN). Areas of a network are usually isolated from each other in areas called District Metered Area (DMA). The entry points may then be regulated for different consumer demands using pressure reduction valves. The most commonly used sensors are pressure and flow, even though several others are available [2]. Pressure in WDN may fluctuate every hour due to the varying consumer demand [2], and an unexpected drop is a indication of an unknown leak or un-documented demand. Features concerning a leak are presence of a leak - leak detection, severity of the leak - leak magnitude, and determining the exact location - leak localization.

A leak produces a decrease in the upstream pressure and downstream flow-rate for a straight pipeline. However, for a complex pipe network, the concept of outlet and inlet pressure may become ambiguous.

An important phenomenon for leak analysis is Negative Pressure Wave (NPW). In theory, a leak will result in an equivalent pressure drop in its vicinity. This gives rise to pressure waves which travel along the pipe in each direction. Its specific velocity corresponds to the leak size.

Leak induced pressure drop may be detectable only for leaks larger than a certain magnitude, but simulations have shown that pressure gradient variations can be recognizable even for smaller leaks (below 1 liter/min) [22]. However, such characteristics are specific to the dimensions of the WDN and the leak size.

Leaks also produce an acoustic signal, and can be detected by various equipments such as acoustic sensors, accelerometers, microphones, and dynamic transducers. In some cases, such as low-temperature regions, a change in temperature can also indicate a leak. These changes can be detected by an optical fiber.

3.6 Generation of Pressure Residuals

Pressure residuals are the underlying philosophy of all model-based leakage analysis [2], and the literature offers several analysis strategies that relies on residuals generated by WDN models, as mentioned in chapter 2. This section describes how the generation of pressure residuals is performed in this project.

The model-based approach aims to detect or locate leaks in a WDN by using pressure residuals made by comparing pressure measurements to estimations obtained by the equivalent location in a WDN model, in a leak-free scenario. The minimum size of a leak is related to the sensor resolution and also the modeling and demand uncertainty. If the WDN is divided into several DMA with known inlet flow, then only a smaller model of a certain DMA needs to be considered instead of the whole WDN [23]. This is because leakage detection can be determined using mass/volume balance for inlet flow and outlet consumption.

Figure 2.5 from chapter 2 shows the scheme of generating residuals. Nodal demands d can be gathered from water meters measuring consumption for each household. leak l_i and noise v are unknown parameters for the real WDN, while boundary conditions c are constraints acting upon the system, such as reservoir pressure and flows. A demand estimation $\hat{d}$ is made based on previous demand data. Global demand d_{WDN} measured at inlet can be used if wireless meters measuring water demand is not available. In addition, the model needs information about pipe diameter, pipe length, pipe roughness, nodal height, pump behaviour and reservoir pressure/head. Estimated pressures $\hat{p}_0$ can then be generated for each time step with a certain resolution where a water demand pattern changes demand over time.

Chapter 4

Design and Specification

This chapter presents a functional specification that describes what the project will accomplish from an outside perspective. A system overview is also provided which shows how our system is respresented in a five-layered architecture.

4.1 Functional Specification

1. The meters and collectors will use the wireless communication protocol wM-Bus with C1 mode. This involves one way communication from the meter to the collector.
2. Two real sensors by Kamstrup will be used. One records flow data and one records pressure data.
3. Six supplementary simulated meters will also be used to send data from other locations. Three sends flow data and the three others send pressure data.
4. The collectors will only receive data packets from specified meters.
5. The wM-Bus data packets will be encrypted.
6. A gateway will receive and send data to a central storage medium.
7. The data packets need to be stored in a way that the measurements are readable.
8. The storage medium will be easily accessible in order to view data at a later time.
9. The collectors need to be able to send and receive data at distances required by a neighbourhood setting.
10. The infrastructure should make analysis of leakages possible.
11. The infrastructure should be able to manage all meters in a timely manner.

4.2 System Overview

In accordance to Antzoulatos G. et al. [8] suggested generic five layered architecture for water management systems, an architecture has been constructed, and is shown in figure 4.1.

End Device Layer handles recording and wireless transmission of pressure and flow data in almost real time. The transmission is done over the wM-Bus protocol in C1 mode every 95 second. The meters are Kamstrup PressureSensor, Kamstrup FlowIQ, and simulated meters implemented using Silicon Labs SLWSTK6220A. They are placed in four different locations.

Device Connectivity Layer handles connection between end devices and a central server in-

frastructure. This is also referred to as a **gateway**. The gateway receives wM-Bus packets by using wM-Bus collectors. In this case, both collectors receive all packets in order to test range capabilities. Each collector is connected to each their own computer through USB. Using a Python script, The two computers read the serial port, formats the M-Bus packets, and sends them to the AWS cloud solution through a wifi connection. the packets are sent using the MQTT message protocol.

Device & Data Management Layer handles reception and storage of metering data. This layer and the next one is hosted by Amazon Web Services (AWS). The IoT core service receives the data through the MQTT protocol. Rules are defined to determine which packets are sent to the Timestream service. The Timestream service handles both short and long term storage. Identity and Access Management (IAM) is used to manage access to cloud resources, while cloudwatch is used to generate logs of various actions performed by the cloud.

Data Processing & Analytics Layer processes available data in the database. In this system, API Gateway manages requests made by an end user. The requests are then sent to Lambda for processing. It needs to retrieve required data from the database. Finally, the result is sent back to API gateway and then to end user. This layer would also be responsible for analytics and preprocessing performed on the available data.

User Interaction Layer represents ways for users to interact with the gathered data. API Gateway is used to request data, while Grafana is used to directly access the Timestream database through the use of a Grafana plugin for Timestream. Grafana can then be used to create dashboards with graphs for visualization.

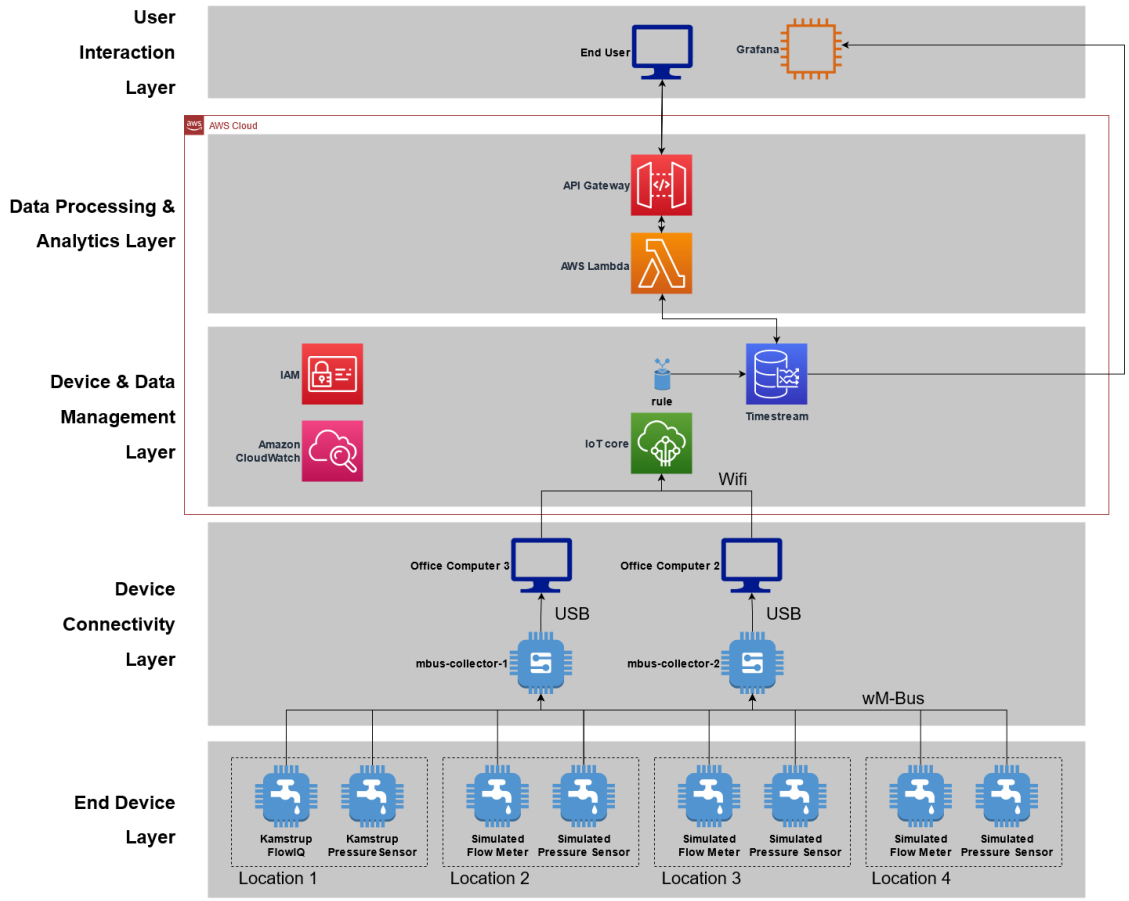


Figure 4.1: An overview of the whole infrastructure. Data Processing & Analytics Layer, and Device & Data Management Layer is contained within AWS cloud solution. The layered architecture was inspired by [8]

Chapter 5

Implementing the Infrastructure

This chapter presents the implementation of the system architecture described in chapter 4 and also how the generation of pressure residuals was performed. The implemented system architecture serves as a proof of concept, and not as a complete solution.

5.1 Configuring Simulated Meters SLWSTK6220A

The simulated meters were implemented using the starter kit SLWSTK6220A by Silicon Labs with a protocol stack by STACKFORCE, which is described in chapter 3. The starter kit is comprised of a radio board with ID BRD4502C and a wireless micro-controller unit with part number EZR32WG330F256G60G. The protocol stack is downloaded from Silicon Labs' own website [24]. Once downloaded, the software provides several different IAR workspaces to choose from. IAR Embedded Workbench is a licence based toolchain that is used to write, compile and upload code to the simulated meters. The specific version used for this project is IAR Embedded Workbench for Arm 8.20.2.

The IAR workspace used in this project is located at `\Wireless M-BUS stack\WMBUS Tools\Firmware\TPL firmw`
This is configured for the starter kit SLWSTK6220A and implements TPL as its top layer, excluding serial interface and APL. This gives more control over how the wM-Bus packets are defined, by making it possible to choose a different header type and change the pre-included timestamp. This was ultimately not done, and the packet structure remained the same.

In order to implement the simulated meter, the correct configurations for the IAR workspace needs to be selected:

1. Open the workspace "`Demo_SLWSTK6220A.eww`", located in mentioned folder, in IAR Embedded Workbench.
2. Make sure configuration `Demo_apptpl` is selected.
3. Select Project → Edit Configurations...
4. A new window will appear. Create a new configuration called "`SLWSTK622A_Meter_C1`", based on `SLWSTK622A_Meter_S2` with the Arm toolchain.
5. Have the root folder selected in the tree viewer
6. Select Project → Options
7. For General Options → Target → Device, it should be set as "SiliconLaboratories EZR32WG330F256R60".

This is the specific part number for this starter kit.
8. For C/C++ Compiler → Preprocessor → Preinclude file, it should be set as "`Meter_C1.h`", located in `\src\configs`.
9. For Debugger → Setup → Driver, it should be set as "`J-Link/J-Trace`"
10. Confirm these settings by clicking OK.
11. Back in the tree view, navigate to location `lib/RF/SI4460C_CortexM4F_Meter_S2/`. The correct RF driver static library "`librf.a`" that is configured for C1 mode needs to be selected. Right-click on the folder and choose Add → Add Files, and select `librf.a` file located in `\libs\rf\SI4460C\CortexM4F\Meter_C1`. The other `librf.a` needs to be excluded from the build by right clicking on it and choosing Options... → Exclude from Build.
12. The same process needs to be performed for "`libstack.a`" located at `/lib/Stack/CortexM4F_Meter_S2_Tpl/` in the treee view. The correct file is located in a folder named `\libs\stack\CortexM4F\Meter_C1\Tpl`. This configures the build to use a wM-Bus protocol that is configured for a meter device in C1 mode with TPL as top layer.
13. The workspace is now correctly configured and `Download and Debug` can be performed with the device connected to the computer through USB. With the default code, the device will start automatically sending packets in C1 mode every 96 second when it is powered on.

Changes are made to the main file `main_meter.c` to make a device send simulation data that is stored in flash on the device. The code file is shown in appendix E. Two separate `main_meter.c` files is used, one for simulated pressure meters, and one for simulated flow meters. Six meters are configured in total.

the following changes were made to `main_meter.c`:

1. two extra header files are included at line 24-25; `em_gpio.h` to configure GPIO/LED pins, and `em_cmu.h` to configure clock signal for peripherals.
2. Identification for each meter is shown in table 5.1, and this is entered on line 107-110. The identification is reentered for each meter that is installed. Encryption key is kept as is.
3. Interval is set to 95 seconds at line 205, and address is set at line 206. This should be set by `wmbus_tpl_start(&gs_start_attr)` at line 209, but doesn't work for some reason.
4. Two LEDs toggle alternately on and off when a packet is sent. Initiation of the LEDs is handled at line 194-198.
5. The toggle of LEDs are performed at line 362-363.
6. Reading of flash and storage as static data is performed at line 236-289. Every packet has 16 bytes of simulated data. This complies to the pressure packet shown in figure. A slight A.1 in appendix E.
7. A slight modification is made for the code of the simulated flow meters. The code section from 236 to 289 is instead replaced with code listing 5.1. this is because wM-Bus packets for flow only contains 11 bytes of measurement data instead of 16. The address jumps 12 bytes instead of 11 for every loop because 4 bytes are writable at a time. This must be taken into account by adding 1 byte of padding to every packet in the flash.
8. A prefix for day and hour is used to choose starting points for the packets loaded onto the flash, as shown on line 146-150. An hour prefix of 12 is used for all meters to start

the recording at the equivalent time of 12:00 during the day. A prefix of either 0, 1 or 2 is added to start reading packets representing Monday, Tuesday and Wednesday. This is done to avoid having the meters broadcast the same values.

```
uint32_t packets = 6117;
uint32_t* start_addr = (uint32_t*)(FLASH_BASE + 2048*16); //location 0x8000
uint32_t* curr_addr = (uint32_t*)(start_addr + curr_packet * 3);

if(curr_packet != packets-1)
  curr_packet++;
else
  curr_packet = 0;

uint32_t sensor_data = *curr_addr;
uint8_t word_part0 = sensor_data >> (0 * BYTE_SIZE);
uint8_t word_part1 = sensor_data >> (1 * BYTE_SIZE);
uint8_t word_part2 = sensor_data >> (2 * BYTE_SIZE);
uint8_t word_part3 = sensor_data >> (3 * BYTE_SIZE);

sensor_data = *(curr_addr + 1);
uint8_t word_part4 = sensor_data >> (0 * BYTE_SIZE);
uint8_t word_part5 = sensor_data >> (1 * BYTE_SIZE);
uint8_t word_part6 = sensor_data >> (2 * BYTE_SIZE);
uint8_t word_part7 = sensor_data >> (3 * BYTE_SIZE);

sensor_data = *(curr_addr + 2);
uint8_t word_part8 = sensor_data >> (0 * BYTE_SIZE);
uint8_t word_part9 = sensor_data >> (1 * BYTE_SIZE);
uint8_t word_part10 = sensor_data >> (2 * BYTE_SIZE);

/* The periodical data includes the time. Further data can be added here
 * to the telegram:
 * Record 0: time information (already set automatically!)
 * Record 1: our example data (pc_staticData[])
 */
uint8_t pc_staticData[]={ word_part0,
                          word_part1,
                          word_part2,
                          word_part3,
                          word_part4,
                          word_part5,
                          word_part6,
                          word_part7,
                          word_part8,
                          word_part9,
                          word_part10};
```

Code listing 5.1: Code modification for simulated flow meters. 11 bytes of data are broadcast every loop, but the address jumps 12 bytes because 4 bytes are writable at a time.

Simulated data also needed to be loaded onto the devices. This was accomplished in the following way:

1. A weeks worth of both pressure and flow data was recorded from the real meters Kamstrup flowIQ and Kamstrup Pressuresensor. Implementation of local storage is described in section 5.2.2.
2. All days of flow data packets and pressure data packets were gathered into each their own csv file. Appendix A shows an overview of the Kamstrup wM-Bus packets, where the blue fields are the measurement data that was gathered. Pressure data contained 6282 pressure packets with a size of 16 bytes, which resulted in a total size of about 98 KiB.

Meter Type	Manufacturer	M-Bus Address	Manufacturer ID	Version	Type	Simulation day
Pressure	Silicon Labs	50902542	ce9a	1d	18	Monday
Flow	Silicon Labs	51705369	ce9a	1d	16	Monday
Pressure	Silicon Labs	51705518	ce9a	1d	18	Tuesday
Flow	Silicon Labs	51705538	ce9a	1d	16	Tuesday
Pressure	Silicon Labs	50902294	ce9a	1d	18	Wednesday
Flow	Silicon Labs	51705516	ce9a	1d	16	Wednesday
Pressure	Kamstrup	77000424	2c2d	01	18	-
Flow	Kamstrup	68826830	2c2d	1d	16	-

Table 5.1: Identification for every meter used in this project. Simulation day is the day the simulated meter starts to simulate first

Flow data contained 6117 packets with a size of 11 bytes. The data was padded to get packets of size 12 bytes, since only 4 bytes are writable at a time. This resulted in a size of about 72 KiB.

3. Flash Programmer by Silicon Labs was used to load the data onto the flash storage of the device. The CSV file would first have to be converted to the IBM HEX format.
4. A custom Python script, called `csv2hex.py` was made based on the intelhex Python library, as shown in appendix C. Both csv files were converted using this script.
5. The simulated meters were loaded with either flow or pressure data on hex format, using the flash programmer. A memory offset of 0x8000 was used to avoid overwrite of the code loaded onto the device. The device's flash size of 256 KiB was more than enough to contain the data. The code size is known because IAR embedded workbench conveys this information when the code is uploaded to the device.

With this, the simulated meters were ready to broadcast encrypted simulated meter data at an interval of 95 seconds. One week of data is contained in the device. When the broadcast of packets reaches the end, it is restarted and packets from the same Monday are broadcast.

Boxes were 3d-printed to protect the circuit, and they were placed at different offices in the same building. Figure 5.1 shows the simulated meters.

5.2 Setting Up the Device Connectivity Layer

5.2.1 WM-Bus Collector

The RC1180 demonstration kit is used as wM-Bus collectors for this project. An overview of the kit is shown in figure 5.2. The RC1180 chip implements the wM-Bus protocol stack, and is closed source. It can be configured by using the UART connection. The UART signal is converted to USB for easy connection to a computer, where a serial port can be used to give commands and receive data. The chip sends and receives wM-Bus packets, as specified by the wM-Bus protocol, using an RF signal. A config button is available that makes the chip enter a configuration mode. A reset button is used to restart the device. To update the device, an interface is available at P8. It is also possible to choose between battery power or power through an USB connection.

A serial port connection is used to configure the device. There are several commands available, and the ones used in this project are shown in table 5.2. Radiocrafts also offers two applications, called MBUS-CCT and MBUS-DEMO, that make the configuration of the device easier.

Figure 5.1: The simulated meters used in the project. 3d-printed boxes were made to protect the circuit while being placed at different locations to broadcast wM-Bus packets. The sides were open so that they could be easily turned on.

This also uses the serial port connection.

The Radiocrafts RC1180-MBUS3-DK is used to implement two collectors that receive and decrypt packages that are broadcasted from wM-Bus meters. The company provides two software tools which make it easier to configure and test the MCU, called MBUS-DEMO and MBUS-CCT. Both can be downloaded as a package called RCTools MBUS on the company's website [26]. This requires an account.

MBUS-CCT gives an interface to the MCU by connecting to a serial port. It enables easy configuration of the module while also providing a terminal window to send and receive data.

MBUS-DEMO can also be used to configure the module, but to a more limited degree since its function is to demonstrate core functionalities. it simplifies the process of demonstrating the different functions of the MCU, such as generating packages, sniffing for packages, and simple configuration. Both of these tools will be used because MBUS-DEMO is quicker to use to test the system, but MBUS-CCT provides all configurations that are necessary for this project.

The collectors were connected to each their own computer with a USB cable to interface with a serial port connection. The computers were close to each other, but one collector, called mbus-collector-2 with address `0x24024024`, had an external dipole antenna to see if it would improve the range. The other collector, called mbus-collector-1 with address `0x30030030`, had a smaller whip antenna which was provided by the development kit.

To receive packets from meters, they need to be installed. Also, the unique encryption key for each meter needs to be added to decrypt the packets. This is done using the terminal window in MBUS-CCT. The following steps are performed to configure the MCU:

1. The MCU is connected to MBUS-CCT using the correct COM with a baud rate of 19200.

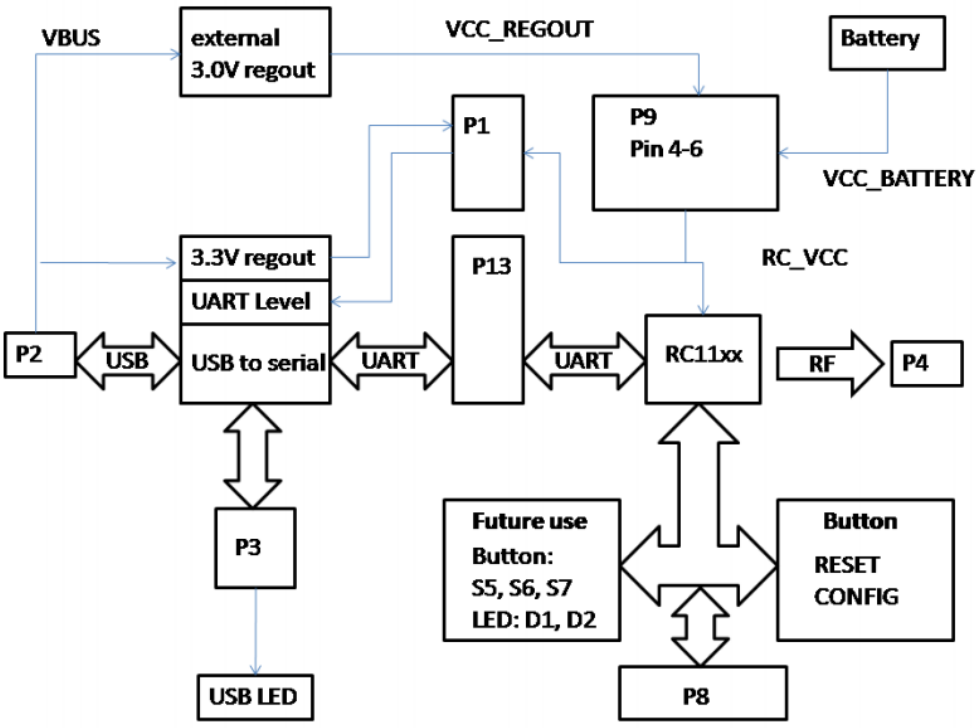


Figure 5.2: Block diagram of the RC1180 demonstration kit. RC11xx is RC1180 for this project. From [25].

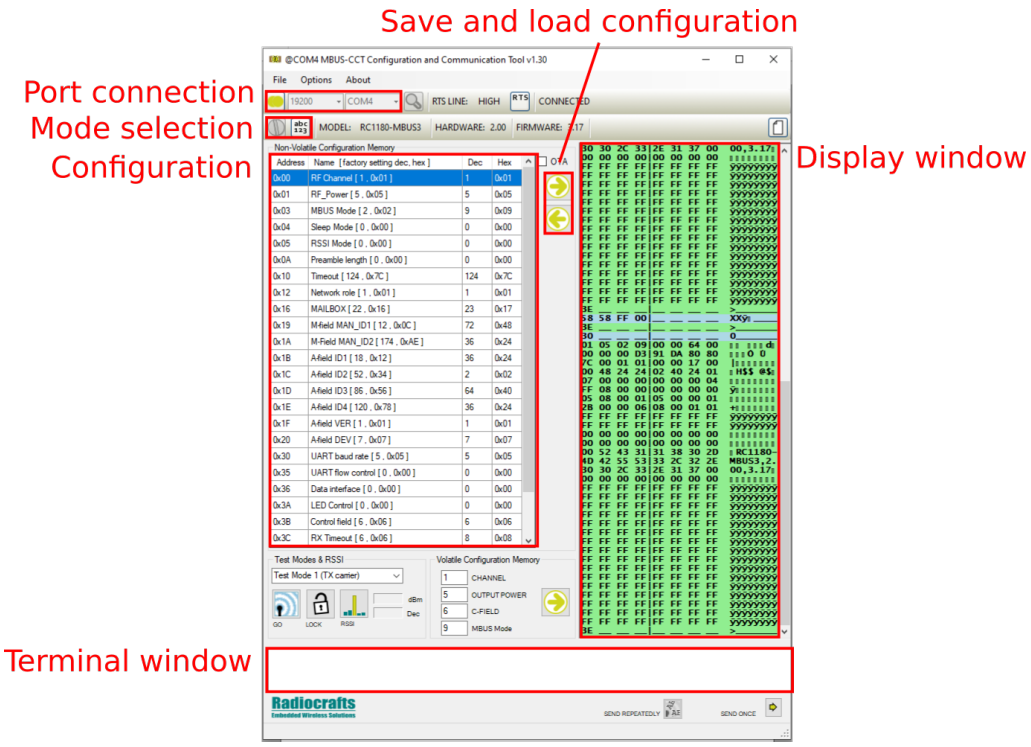


Figure 5.3: Overview of MBUS-CCT program window

2. Configuration mode needs to be entered by pressing the button "Enter configuration mode". Sometimes the configuration button on the MCU needs to be pressed right afterwards.
3. Afterwards the default configuration can be loaded by pressing "Load configuration from the modules non-volatile memory".
4. The configurations in table 5.2 are used. These are entered into the configuration window shown in figure 5.3, where each address is associated with a value.
5. After changes, save them by pressing "Send configuration to the non-volatile memory".
6. Enter terminal mode by pressing its button at "Mode selection" in figure 5.3.
7. The commands listed in code listing 5.2 is used in chronological order to install the various meters.
8. The collector is now configured and ready to be used. When connected through USB to a serial port on a computer, it will automatically send packets it receives to the port.

```
$ B
$ '1 0x24 0x04 0x00 0x77 0x2D 0x2C 0x01 0x18'
$ K
$ '1 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX'
$ B
$ '2 0x30 0x68 0x82 0x68 0x2D 0x2C 0x1d 0x16'
$ K
$ '2 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX'
$ B
$ '3 0x42 0x25 0x90 0x50 0x9a 0xce 0x1d 0x18'
$ K
$ '3 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
$ B
$ '4 0x69 0x53 0x70 0x51 0x9a 0xce 0x1d 0x16'
$ K
$ '4 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
$ B
$ '5 0x18 0x55 0x70 0x51 0x9a 0xce 0x1d 0x18'
$ K
$ '5 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
$ B
$ '6 0x38 0x55 0x70 0x51 0x9a 0xce 0x1d 0x16'
$ K
$ '6 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
$ B
$ '7 0x94 0x22 0x90 0x50 0x9a 0xce 0x1d 0x18'
$ K
$ '7 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
$ B
$ '8 0x16 0x55 0x70 0x51 0x9a 0xce 0x1d 0x16'
$ K
$ '8 0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF'
```

Code listing 5.2: Terminal commands used to install all meters for the collectors. Both collectors receive all data packets in order to test signal strength. Encryption key for real meters are not shown because of security considerations.

The 'B' command is used to bind a meter to an address. The 'K' command is used to specify a unique key for the meter. The first number determines the address location. For binding, the unique address, manufacturer address, version and type needs to be provided. Numbers need to be enclosed in apostrophe to translate them as numbers and not ASCII symbols. The MCU can store 64 meters in total. For every command, the terminal window will respond with $3E$, or '>' in ASCII, to verify that the command was received.

Address	Name	Hex Value	Description
0x03	MBUS Mode	0x09	Set M-Bus mode as C1
0x05	RSSI Mode	0x01	Enable logging of RSSI value for packets
0x12	Network Role	0x01	Set the device's network role as collector
0x1B	A-field ID1	0x24	Set first byte of unique ID to 0x24
0x1C	A-field ID2	0x02	Set second byte of unique ID to 0x02
0x1D	A-field ID3	0x40	Set third byte of unique ID to 0x40
0x1E	A-field ID4	0x24	Set fourth byte of unique ID to 0x24
0x3D	Install Mode	0x00	Set the device's install mode as normal
0x3E	Encrypt Flag	0x01	Enable encryption for device
0x3F	Decrypt Flag	0x01	Enable decryption for device

Table 5.2: Configurations for wM-Bus collector. Other settings are kept as default. This sets it to use C1 mode, add RSSI value for packets, behave like a master/collector, sets a unique ID, operate in normal mode and enable encrypt/decrypt. Normal mode means that only data packets from installed meters are received.

5.2.2 Serial Port Logger

A Python script was created to perform several tasks, and is shown in appendix B. It listens to a certain port where the wM-Bus collector transmits its received wM-Bus packets from the installed meters. The received packets are formatted and stored locally on the office computer in both its formatted and original state. The formatted packet is finally converted into a JSON package to be sent to the cloud solution using the publish-subscribe protocol MQTT, which runs over the TCP/UDP protocol.

MQTT implements a publish/subscribe pattern, instead of wM-Bus' client-server model. In between there is a MQTT broker which filters incoming messages based on **topics** defined by publisher, and distributes them to the correct subscriber. This model makes subscriber and publisher decoupled as they don't have to be aware of each other to communicate. For this architecture, the IoT Core implements the broker.

The program starts from its main function on line 47. When a wM-Bus collector is connected through a serial port, `print_ports()` can be used to print out connected ports and find the correct on. Its functionality, along with other serial port manipulations, are provided by the serial library.

When the correct port is found, it can be logged by using `log_ports()` on line 294. The whole received packet is read using a for-loop. The time it was received is saved.

The packet is then formatted using `format_packet(pac)` on line 181. This is done, according to the wM-Bus standard, to make the values of the received packet understandable. Each packet is identified by its manufacturer ID and packet type, as shown on line 199-239. The real meters by Kamstrup transmits two types of packets, which are shown in figure A, where DIF and VIF information are only transmitted every 10 packet. This information is stored in a dictionary at line 33 for every meter and used for the subsequent packets, before being updated.

Both formatted packets and the raw wM-Bus packets are saved locally on the computer. The formatted data is saved to CSV files with a file name containing the location of the data and the current date. The location of every meter is shown in the infrastructure overview in figure

4.1. Raw data is saved to CSV files with a file name containing the device ID and the current date.

After saving formatted and original packet locally in a CSV format, the formatted packet data is sent to the cloud solution. The data to send is extracted by the formatted packet, as shown on line 370-390. This is converted into a json format on line 395. A topic for the MQTT protocol is defined at line 394. An effort is then made to publish the JSON package to the defined topic through the MQTT client, as shown on line 397. The MQTT client is implemented using the library called `AWSIoTPythonSDK`. It is initialized at line 298-300. This requires key files from the IoT Core service, which is covered in next section. All error information is logged in a csv file with a timestamp, as shown on line 403-409, before continuing.

5.3 The Cloud Solution

The cloud architecture was implemented using a Amazon Web Services (AWS) cloud solution. The reason for this choice was because AWS is one of the oldest and most popular cloud service providers at the time of writing. This service implements the layers Device & Data Management Layer, and Data Processing & Analytics Layer of the infrastructure shown in figure 4.1. AWS provides services to be used as building blocks to build up a cloud infrastructure. Several services are available and many can be used to achieve the same purpose. The ones used in this project can be viewed in figure 4.1. How they are implemented is explained in this section.

AWS requires an account to be set up with a card payment method tied to it. The account is then charged for the amount of usage of each service. The specific unit of measurement for usage depends on each service. The cloud infrastructure is managed through their website on `https://eu-west-1.console.aws.amazon.com`. It is possible to choose different locations all over the world for the deployment of the cloud solution. For this project, the location `eu-west-1` was chosen, which is located in Ireland. There are closer servers, for example in Sweden, but since Ireland is one of the oldest it provides much more features than most others. Timestream is one such feature only available at certain servers.

5.3.1 Identity and Access Management (IAM)

IAM is used extensively by AWS cloud to control the access to resources. User identities are created to give permission to access certain resources. In this project, only one user identity with administrator/unlimited access to the cloud was created. Roles are similar to users, only that it is not associated to a certain entity, but rather granted to trusted users, applications or services. Every service in the cloud has a role granted for it to access the required resources. Policies defines permissions in a JSON format, and are attached to roles and users. It is possible to create custom policies, but there are often default policies available to use. A policy specifies if an action is allowed or denied, specifies which action is being allowed or denied, and specifies on which resource the action is allowed or denied

5.3.2 Cloudwatch

Amazon Cloudwatch is used to monitor the various services in the cloud in near real time. This is in the form of logs, events and metrics that are defined by the user. This can be used to create alarms that activate when certain thresholds are reached. It can be used to troubleshoot services to see if they function as expected. The metrics can also be used by other services to

trigger logic to execute on certain conditions. For this project, it was used to troubleshoot the API gateway in order to see if the correct events were transmitted by a lambda function.

5.3.3 IoT Core and Rules

IoT Core acts as a message broker for the MQTT protocol for this system. Messages are published on a certain topic, and IoT Core transmits these to a certain destination based on rules. This way, IoT Core acts as a communication interface to and from the cloud solution. Using so called rules, simple SQL queries can be created to transfer messages to other cloud services in AWS. The IoT Core keeps a list of things, which is a representation of a connected device in the cloud.

A **thing** has to be defined to setup communication with the gateway and IoT Core. The thing represents a physical device in the AWS cloud. At the home page of IoT Core, a **thing** was created with the name mbus-collector-1. Upon creation, a IoT certificate, public key and private key was generated. These were used as files to give the Python script for serial logging the necessary authentication to upload its content to the cloud. The Python script also needed a CA certificate for server authentication, which is provided from the AWS web page. The generated IoT certificate needs a IoT policy attached to have the necessary permission to allow a device to publish its content to a topic using MQTT. These IoT policies are similar to the IAM policy, but act upon devices connecting to the cloud instead. With these configurations, the Python script should be able to publish its content to the IoT core on a certain topic. The same steps had to be done for the second collector, called mbus-collector-2, as well.

A rule was set up by going to Act → Rules for IoT core, which sent messages to the Timestream database. Its name was `iot_timestream`. Using SQL, a query statement was created:

```
1 SELECT min_pressure, max_pressure, inst_pressure, flow_inst, flow_max_month, flow_inst_diff,
      flow_max_month_diff, temp_ambient, RSSI
2 FROM 'collectors/#'
```

This defines which attributes are to be extracted from a topic source. # is a wild card statement which selects every attribute. Then an action was added to the rule. The pre-defined action "Write a message into a Timestream table" was chosen. The settings shown in figure 5.4 were used. This sets the desired database and table, which are described in the Timestream section. Timestamp is added automatically based on time of upload, while dimensions represent metadata for each measurement value. The structure of the data is explained further in next section.

5.3.4 Timestream

Amazon Timestream is a scalable database that stores time series data. It consists of several layers, as shown in figure 5.5. The reason for choosing Timestream is because it is optimized to store data for several IoT devices. According to Amazon, when storing and analyzing trillions of events per day, it is 1000 times faster and 1/10th the cost compared to their other services providing relational databases, such as the popular DynamoDB. It provides an easy interface both for data ingestion and extraction with querying. Also, a solution for long-term storage is implemented, using magnetic storage. The database is referred to as **serverless**, since scaling of data ingestion, storage and querying is handled automatically without intervention as the system grows.

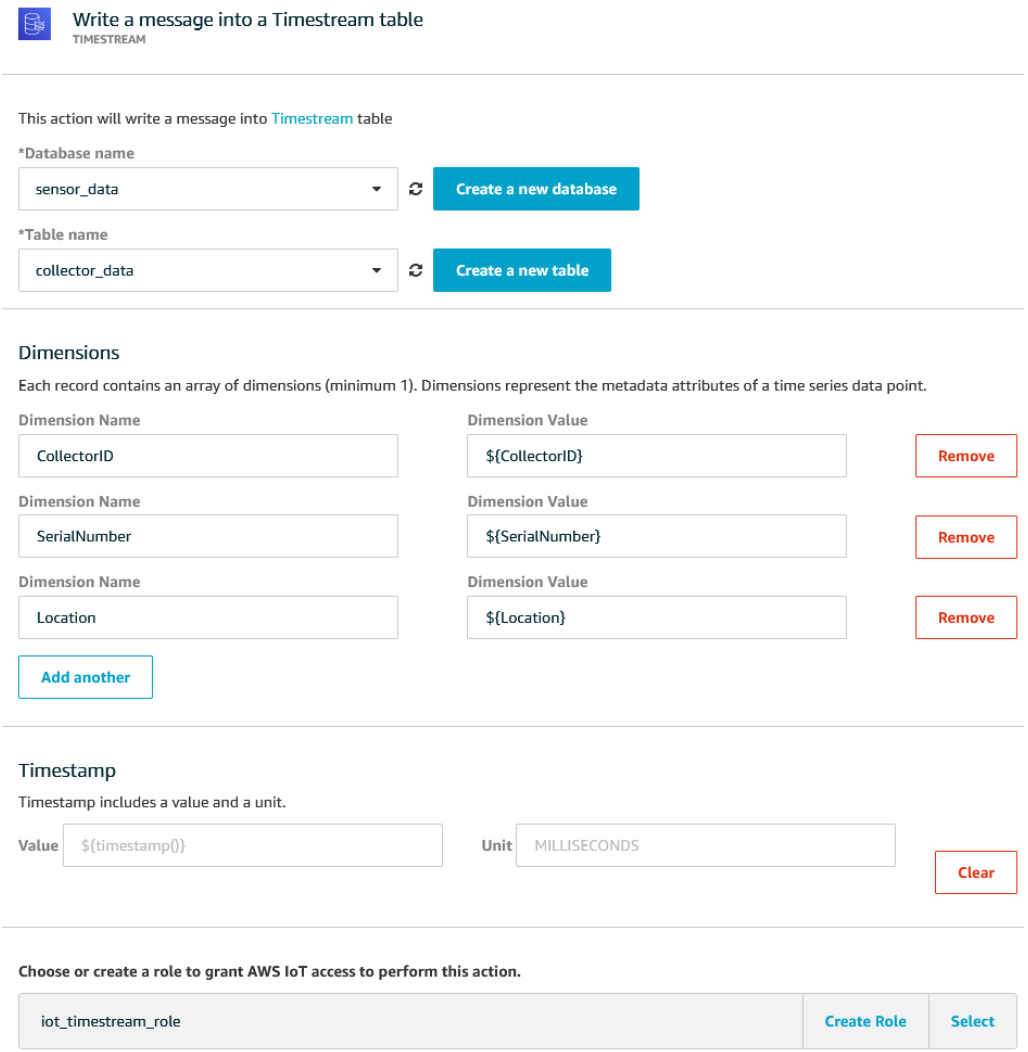


Figure 5.4: Settings used for action associated with the AWS IoT rule. A created Timestream database with a table is chosen. Dimensions are metadata that is added to every record in the Timestream table. A new timestamp is added automatically. A role is added with the necessary permissions to access the Timestream database.

measure_name	**measure_type**
flow_inst_diff	measure_value::bigint
flow_max_month_diff	measure_value::bigint
temp_ambient	measure_value::bigint
RSSI	measure_value::bigint
min_pressure	measure_value::double
max_pressure	measure_value::double
inst_pressure	measure_value::double
flow_inst	measure_value::double
flow_max_month	measure_value::double

Table 5.3: All possible values for measure_name including its type in a record for Timestream series. While one type is chosen, the other one has no value.

Data ingestion is handled by using SDKs for different programming languages or the service's API, which is done by the IoT rule in the former section. Data is written to and automatically indexed in an in-memory storage. After a certain defined time duration, the data is transferred to magnetic storage for long-term storage. The data is deleted after a certain amount of defined time.

From here, the data is available through calls from SQL queries. Since the data is encrypted, access needs to be granted to applications wanting to make a query.

Timestream stores data in a **vertical** fashion, which means every record only has one measurement value tied to it. The data is stored in the convention shown in figure 5.6. The measurement value of the record is defined by `measure_name`. The different values of `measure_name` are shown in table 5.3.

A new Timestream database was created with the name `sensor_data`. The "Standard database" configuration was used. A new encryption key was created and set as a Master key. Afterwards a new table was created with the name "collector_data". The table was attached to the database and parameters for data retention were set. For memory store retention, 1 month (30 days, 11 hours) was used. For magnetic store retention, 2 years were used. The Timestream database is now set up.

5.3.5 API Gateway and Lambda

API Gateway was used to make third-party applications able to access data inside the cloud solution. The type of API used was based on REpresentational State Transfer (REST), which implements HTTP methods such as GET, POST, PUT, PATCH and DELETE. The REST API is made up of the mentioned methods along with resources to act upon. This results in actions that either manipulate the resources or produce a response. For this project, these API responses could be used for an application to create graphs for analysis. The API Gateway sent its requests to a lambda function where it was processed and returned as a result. API Gateway then returned a response to the client that made a request.

Lambda is a service by AWS to run code snippets, called lambda functions. There are a wide variety of available programming languages, but this project uses Python. The code snippet, as shown in code listing 5.3, ran every time the API Gateway made a call to it. To get an API

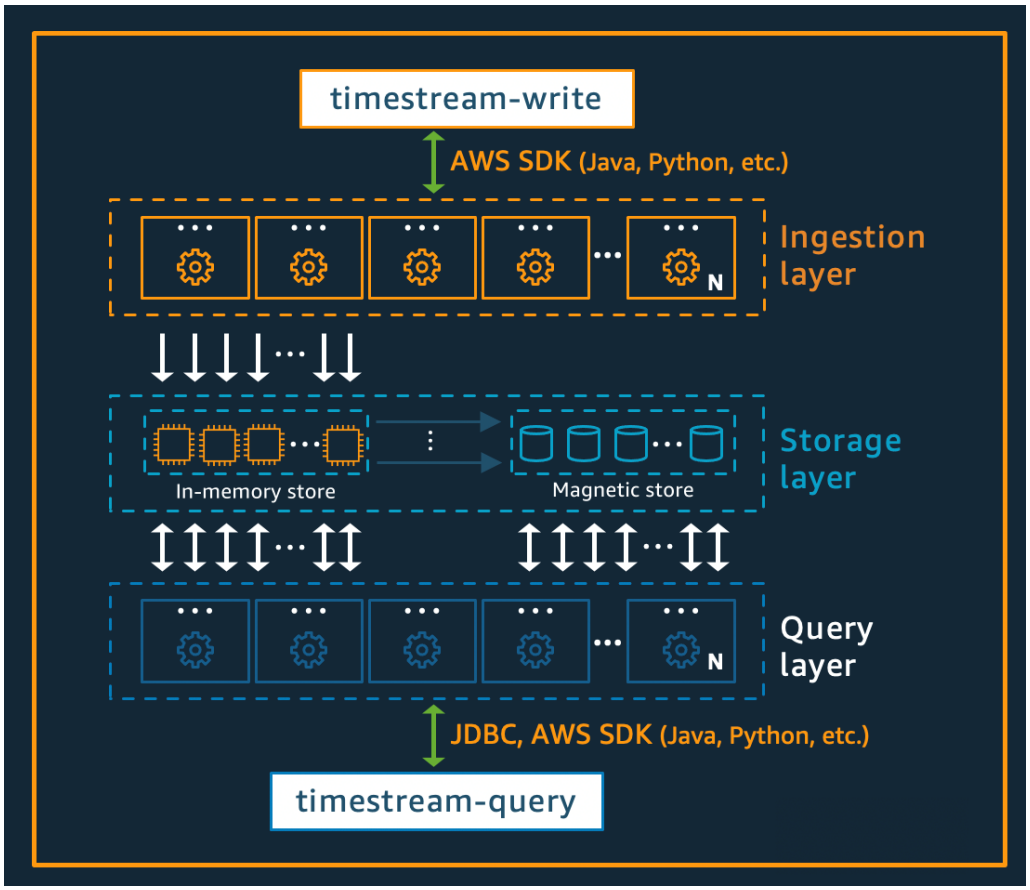


Figure 5.5: The layered architecture of Amazon Timestream. From [27].

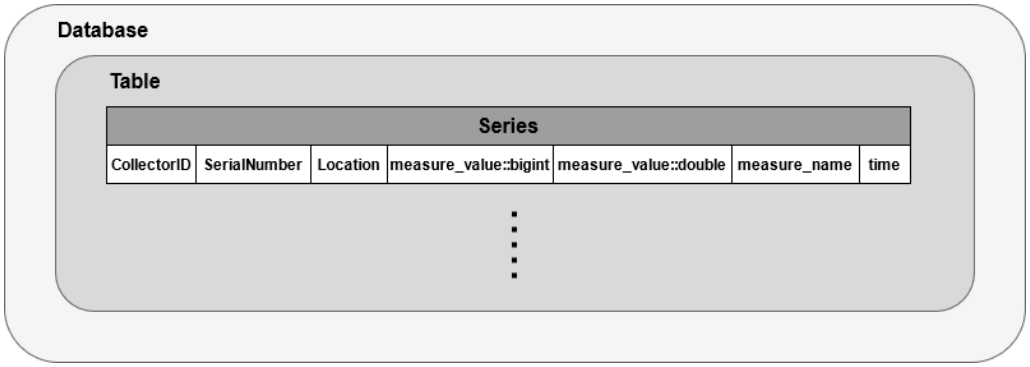


Figure 5.6: Overview of how a record is stored in the database. Inspired by [28].

to access AWS services, the SDK for Python called Boto3 was used. By using this SDK it was possible to query the Timestream database for data.

```
1  import json
2  import boto3
3
4  def lambda_handler(event, context):
5      #print("Got event\n" + json.dumps(event, indent=2))
6
7      client = boto3.client('timestream-query')
8
9      # Static Variables
10     select = "*"
11     database = "sensor_data"
12     table = "collector_data"
13     time_start = "ago(15M)"
14     time_finish = "now()"
15
16     # Perform timestream Query
17     query_string = f"""
18         SELECT {select}
19         FROM "{database}"."{table}"
20         WHERE time between {time_start} and {time_finish}
21         ORDER BY time DESC LIMIT 10
22         """
23
24     response = client.query(
25         QueryString=query_string,
26         MaxRows=10
27     )
28
29     # Construct the body of the response object
30     sensorResponse = response
31
32     # Construct http response object
33     responseObject = {}
34     responseObject['statusCode'] = 200
35     responseObject['headers'] = {}
36     responseObject['headers']['Content-Type'] = 'application/json'
37     responseObject['body'] = json.dumps(sensorResponse)
38
39     return responseObject
```

Code listing 5.3: Lambda function to process requests from the API Gateway. The function structures a query to send to Timestream, which is then returned as a response.

A new API was created through the API Gateway service, with REST being chosen as its protocol. The Endpoint type was chosen to be regional. The name of the API was "wdn-api". A new method called GET was implemented by using the action "Create Method". Lambda function was chosen as its integration point with name of the function server location eu_west_1 provided. Lambda proxy integration was used so that details of the event are available to the lambda handler.

A Lambda function was created with the name "sensor_data_processor". Its content is shown in code listing 5.3. The code performed a query to the database, as shown on line 16-22. A response was created and finally returned. This is a simple GET request to demonstrate the functionality. Further methods could be implemented as well as more detailed responses.

The API could then be deployed to a stage. This was done by navigating to the API Gateway main page and choosing Actions → Deploy API. The API was then deployed to a stage which

Figure 5.7: Map of the nearby water grid of the water meters.

was called "test". The API was then avaiable at:
`https://doqhob3z60.execute-api.eu-west-1.amazonaws.com/test`.

5.4 Creating the EPANET Model

An EPANET model was constructed based on a small part of the water grid layout around the office building where the system of meters was located. This layout is pictured in figure 5.7. On the EPANET software, in Project → Defaults → Hydraulics, flow units were set to Litres per second (LPS) and headloss formula to Hazen-Williams (H-W).

The resulting created model is shown in figure 5.8. Each node $[J2, ..., J9]$ represents a place where there is an outlet to tap water. The meter for this project was thought to be located at node $J5$. As no information of outlet locations were provided, these are made up locations for demonstration purposes. $J3$, $J4$, $J5$, $J7$ and $J8$ have a demand pattern that is found by measurements of flow from the flow meter at node $J5$. The demand pattern was created from the arbitrary date of 2$^{\text{nd}}$ of march, 2021. This is explained in the next section. $J7$ is assumed to have a multiplier of 2 for this demand pattern. $J1$ is assumed to be a reservoir location with constant nodal head that provides a constant flow rate. Its total head is 60.76, which is based on average pressure between 00:00 and 03:00 for the previously mentioned date. This time range is assumed to have little to no consumption which affects the pressure in the WDN. The pipe sections $[a, ..., g]$ have properties that are listed in table 5.4.

The model can also be expressed as a system of equations by using the theory described in chapter 3.4.3. The diagonal Jacobian matrix $\mathbf{D_{11}}$ is shown in equation 5.1. The elements are defined in equation 3.15.

$$\mathbf{D_{11}} = \begin{bmatrix} D_a & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & D_b & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & D_c & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D_d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & D_e & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & D_f & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & D_g \end{bmatrix} \tag{5.1}$$

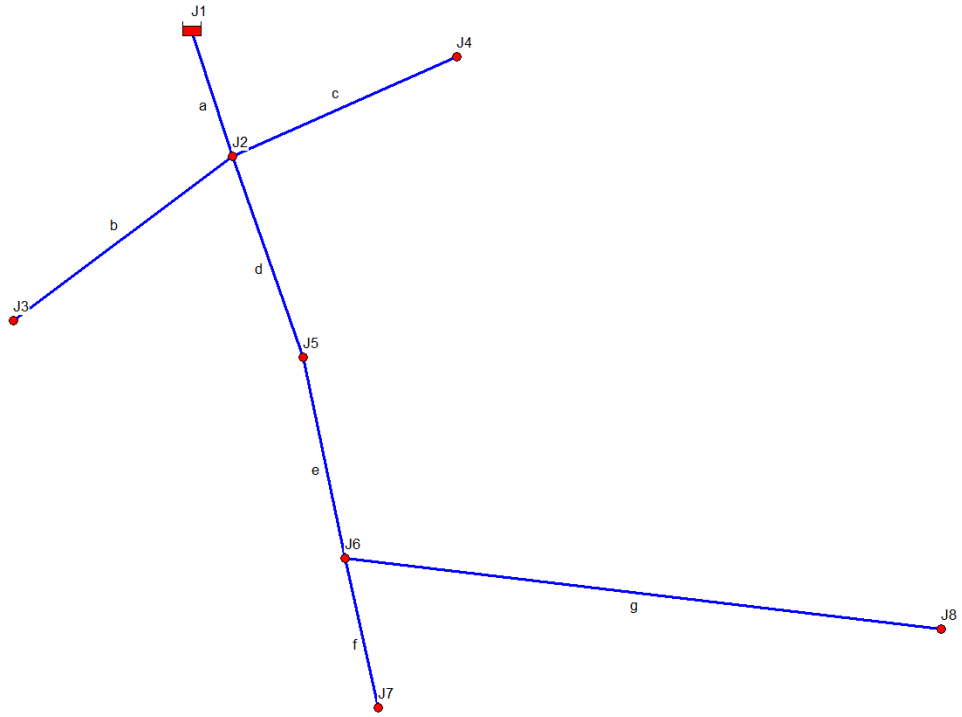


Figure 5.8: Resulting EPANET model of the nearby water grid of the water meters. J1 is assumed to be a reservoir with constant head.

Pipe Section ID	Length L [m]	Diameter d [mm]	Hazen Coefficient C
a	52.34	150	100
b	107.13	150	100
c	80.24	150	100
d	58.39	150	100
e	53.60	150	100
f	40.02	150	100
g	188.71	150	100

Table 5.4: Parameters used for EPANET model for every pipe section. The height of nodes is kept at zero.

The pipes connecting unknown head nodes are expressed as following:

$$\mathbf{A_{12}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{5.2}$$

All iteration step differences for the model are:

$$\mathbf{dQ} = \begin{bmatrix} dQ_a \\ dQ_b \\ dQ_c \\ dQ_d \\ dQ_e \\ dQ_f \\ dQ_g \end{bmatrix}, \mathbf{dQ} = \begin{bmatrix} dH_2 \\ dH_3 \\ dH_4 \\ dH_5 \\ dH_6 \\ dH_7 \\ dH_8 \end{bmatrix} \tag{5.3}$$

The expressions that indicate how far from zero a guess for pipe flows **Q** and nodal heads **H** are, is shown as:

$$\mathbf{F_1} = \begin{bmatrix} R_a Q_a^{1.852} + H_2 - H_1 \\ R_b Q_b^{1.852} + H_3 - H_2 \\ R_c Q_c^{1.852} + H_4 - H_2 \\ R_d Q_d^{1.852} + H_5 - H_2 \\ R_e Q_e^{1.852} + H_6 - H_5 \\ R_f Q_f^{1.852} + H_7 - H_6 \\ R_g Q_g^{1.852} + H_8 - H_6 \end{bmatrix}, \mathbf{F_2} = \begin{bmatrix} Q_a - Q_b - Q_c - Q_d - Q_2 \\ Q_b - Q_3 \\ Q_c - Q_4 \\ Q_d - Q_e - Q_5 \\ Q_e - Q_f - Q_g - Q_6 \\ Q_f - Q_7 \\ Q_g - Q_8 \end{bmatrix} \tag{5.4}$$

These are all required equations to perform a step-wise iteration to a similar solution as the one EPANET produces. This needs to be done for every water consumption in the water demand pattern.

5.4.1 Creating the Demand Pattern

The water demand pattern is created from real volume readings from 2nd of march, 2021. A time resolution of 10 minutes is used, which means averaging out the volume readings that are made every 95 seconds to get a more accurate measurement. Since the demand pattern needs to have a unit of litres per second (LPS), the volume reading in cubic metres also needs to be divided based on time range. A Python script for data analysis in appendix D generates the water demand pattern. It upsamples volume readings to every one minute, before down-sampling to every ten minutes, as shown in function `get_interpolated_flow`. Then, the value is multiplied on line 51 in `get_flow_diff`. First with 1000/600 to convert for cubic meter to litres per second, and then with a multiplier of 120 to have the pressure values of the WDN model match with the real pressure measurements.

The resulting demand pattern, generated from the analysis script, is shown as a red line in figure 5.9, with the volume readings in litres as comparison. This demand pattern is then used for the EPANET model to generate estimated pressure values to be used for pressure residuals. Figure 5.10 shows the demand pattern editor with these values as input.

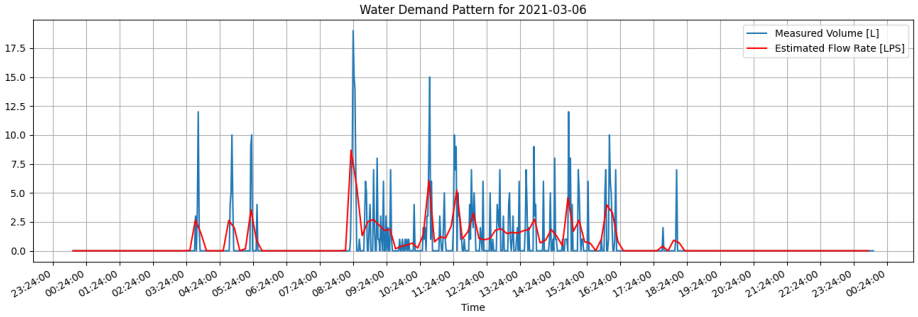


Figure 5.9: The water demand pattern in red based on real volume readings shown in blue.

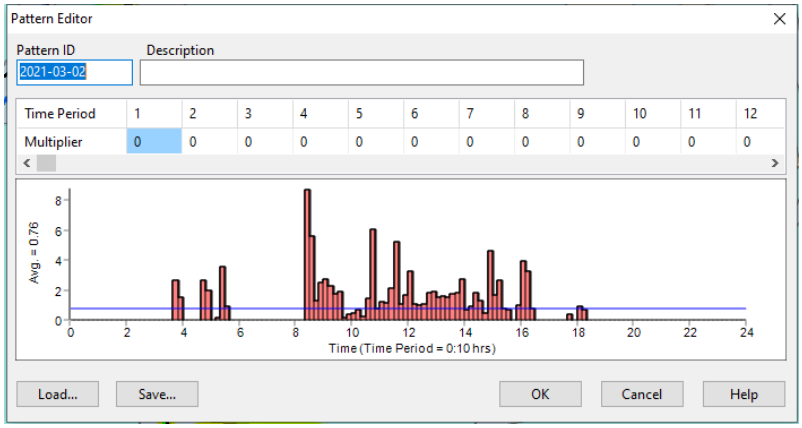


Figure 5.10: The water demand pattern as inputted into EPANET.

CollectorID	SerialNumber	Location	measure_value::bigint	measure_value::double	measure_name	time
mbus-collector-1	770004242c2d	loc-1	-	5.6	inst_pressure	2021-05-21 17:01:58
mbus-collector-1	770004242c2d	loc-1	-	5.61	max_pressure	2021-05-21 17:01:58
mbus-collector-1	770004242c2d	loc-1	-	5.6	min_pressure	2021-05-21 17:01:58
mbus-collector-1	770004242c2d	loc-1	120	-	RSSI	2021-05-21 17:01:58
mbus-collector-1	50902294ce9a	loc-1	-	5.63	inst_pressure	2021-05-21 17:01:19
mbus-collector-1	50902294ce9a	loc-1	-	5.64	max_pressure	2021-05-21 17:01:19
mbus-collector-1	50902294ce9a	loc-1	-	5.63	min_pressure	2021-05-21 17:01:19
mbus-collector-1	50902294ce9a	loc-1	86	-	RSSI	2021-05-21 17:01:19

time	CollectorID	SerialNumber	Location	inst_pressure	max_pressure	min_pressure	flow_inst_diff	temp_ambient	RSSI
2021-05-21 17:01:58	mbus-collector-1	770004242c2d	loc-1	5.6	5.61	5.6	-	-	120
2021-05-21 17:01:19	mbus-collector-1	50902294ce9a	loc-1	5.63	5.64	5.63	-	-	86

Table 5.5: Visualization of how the Timestream record structure needs to be changed in order to be used by Grafana panels. Top is the Timestream records and bottom is after changes.

5.5 Data Visualization and Analysis

5.5.1 Visualization Using Grafana

Grafana is a web application by Grafana Labs to visualize and analyze data. For this project, the Timestream database was connected, so that Grafana can make query calls to update graphs that are created by the user. A collection of graphs is called a Dashboard. The look of the Dashboard is saved so that it can be accessed at a later time.

To connect to the Timestream database, a Grafana plugin called "Amazon Timestream" was used. The plugin performs queries to the Timestream database to gather data. Since the data is encrypted, an authentication provider is required. The administrator user created with IAM is used for this purpose. Through this user, a access key ID and secret access key is provided. In addition, the region, name of database and name of table is provided. The configuration is then saved.

A new dashboard is then created with the name "WDN Dashboard". Inside the dashboard, several panels are created and will be presented in chapter 6. Each panel uses its own SQL query. A challenge of querying the Timestream database, is that the table is vertical as it presents a measurement value for each row. A Grafana graph would like to have the data presented horizontally with all data points for a certain time for each row. The problem is visualized in table 5.5. This is done through the SQL query that is used to get the data. The general structure of a query is shown in code listing 5.4. The query creates a new variable for every value that is needed based on different criteria. This way, The vertical table can be converted to a horizontal one. The Grafana Dashboard could then be used to visualize the required data.

```
1  SELECT time, CollectorID, SerialNumber,
2    MAX(CASE WHEN measure_name = 'inst_pressure' THEN "measure_value::double" END) AS
       inst_pressure,
3    MAX(CASE WHEN measure_name = 'max_pressure' THEN "measure_value::double" END) AS max_pressure,
4    MAX(CASE WHEN measure_name = 'min_pressure' THEN "measure_value::double" END) AS min_pressure,
5    MAX(CASE WHEN measure_name = 'flow_inst_diff' THEN "measure_value::bigint" END) AS
       flow_inst_diff,
6    MAX(CASE WHEN measure_name = 'temp_ambient' THEN "measure_value::bigint" END) AS temp_ambient,
7    MAX(CASE WHEN measure_name = 'RSSI' THEN "measure_value::bigint" END) AS RSSI
8  FROM
9    $__database.$__table
10 GROUP BY
11   time, CollectorID, SerialNumber
12 ORDER BY
13   time desc
```

Code listing 5.4: A standard structure of a SQL query used by Grafana to get data from the Timestream database.

5.5.2 Generating Pressure Residuals

To generate the pressure residuals, data is extracted from the created model and the real measurements. To extract data from the EPANET model, it is first run. Then pressure/head values for node $J5$ is used because this represents the location of the real water meters used in this project. Select the node and click Table → Time series for node $J5$. Then, the data for Head (m) can be selected and copied into an excel document. The values will have to be converted into bar from meter head by using a multiplier of 0.0980.

The real measurements for the equivalent date of 2^{nd} of march, 2021 is used. This data is gathered from the local files generated by the serial port logger code running on an office computer. The data needs to be resampled into values every 10 minute. The Python script for data analysis in appendix D is used for this purpose. The resulting data is saved to an excel document and can then be used. The difference between these two datasets is then used to create pressure residuals.

Chapter 6

Testing and Results

With the various components of the infrastructure implemented, it could then be tested out according to its specification. With the infrastructure's layered design, the various layers were tested in a bottom-up approach. The End Device Layer was tested first and then ending with the User Interaction Layer and then analysis of the resulting data. The serial logging code, as shown in appendix B, was made in such a way that it could test out various parts step wise.

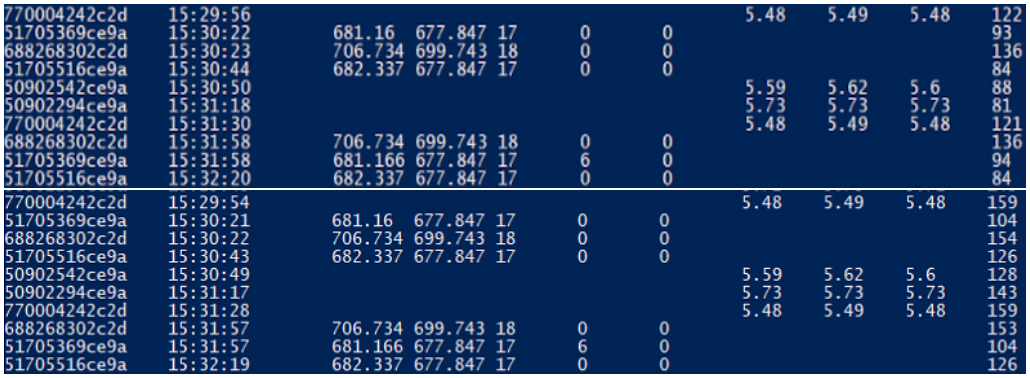
6.1 Receiving Correct Data and formatting Packets in the Device Connectivity Layer

The system was first tested to see if the wM-Bus collector was able to receive packets from the two real meters. This was done by checking if the serial port logging code was able to read any raw wM-Bus packets on the serial port and save them to a local output file. The collector and computer were placed in the same room as the meters to assure they were in range wirelessly. Part of a resulting output file is shown in figure 6.1. The file saved the wM-Bus packet alongside seconds after midnight on the first column, as specified by the code. The result also shows that the packets are decrypted correctly, since column 'U' gives the correct Control Information (CI) of 78 and 79 for Kamstrup flowIQ packets. The values on the last column shows RSSI values, as configured for the collector.

As the interface between End Device Layer and Device Connectivity Layer was working properly, the simulated meters could also be tested. To make sure that the meters themselves were

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU	AV	AW	AX	AY	AZ	BA
1	55796	28	44	2d	2c	24	4	0	77	1	18	8d	20	95	84	68	c1	21	69	2f	79	de	73	3f	14	24	2	25	2	24	2	35	18	5b	39	8d	3d	bf	b8	0	0	7a												
2	55890	28	44	2d	2c	24	4	0	77	1	18	8d	20	96	a8	68	c1	21	97	e1	79	de	73	12	9c	24	2	25	2	24	2	3b	b1	6c	39	a2	e2	8a	39	0	0	79												
3	55984	28	44	2d	2c	24	4	0	77	1	18	8d	20	97	bc	68	c1	21	52	3a	79	de	73	0a	f1	24	2	24	2	24	2	67	d3	62	39	9d	36	69	b9	0	0	78												
4	56078	28	44	2d	2c	24	4	0	77	1	18	8d	20	98	d0	68	c1	21	95	ae	79	de	73	35	4a	24	2	24	2	24	2	c3	cd	80	39	8c	f5	60	39	0	0	7a												
5	56172	33	44	2d	2c	24	4	0	77	1	18	8d	20	99	f4	68	c1	21	fe	1b	78	22	69	24	2	12	69	24	2	2	69	24	2	5	ff	9	13	e1	4a	39	5	ff	0a	89	e0	60	b9	2	fd	17	0	0	79
6	56266	28	44	2d	2c	24	4	0	77	1	18	8d	20	9a	8	69	c1	21	a2	0	79	de	73	6b	85	29	2	2a	2	2a	2	44	b5	cb	39	85	d6	46	39	0	0	79												
7	56360	28	44	2d	2c	24	4	0	77	1	18	8d	20	9b	2c	69	c1	21	ad	22	79	de	73	63	85	2b	2	2b	2	2b	2	e9	2	5b	39	c2	db	4e	b9	0	0	79												
8	56454	28	44	2d	2c	24	4	0	77	1	18	8d	20	9c	30	69	c1	21	94	b2	79	de	73	23	1e	2c	2	2d	2	2c	2	7f	a7	82	39	b1	3a	94	39	0	0	7a												
9	56549	28	44	2d	2c	24	4	0	77	1	18	8d	28	9d	54	69	c1	21	bc	d2	79	de	73	ae	19	df	1	35	2	1a	2	9c	a4	4c	3d	87	d6	48	bd	6	0	7a												
10	56643	28	44	2d	2c	24	4	0	77	1	18	8d	20	9e	68	69	c1	21	6d	d8	79	de	73	d8	ca	2d	2	2e	2	2e	2	15	33	7d	39	9d	fc	47	39	0	0	79												

Figure 6.1: Raw wM-Bus packets saved to a local file using the serial port logging code of appendix B. In addition, a "seconds after midnight" parameter is added upon reading which is saved for each packet.



```
770004242c2d  15:29:56                                          5.48  5.49  5.48  122
51705369ce9a  15:30:22  681.16  677.847 17   0     0                              93
688268302c2d  15:30:23  706.734 699.743 18   0     0                              136
51705516ce9a  15:30:44  682.337 677.847 17   0     0                              84
50902542ce9a  15:30:50                                    5.59  5.62  5.6   88
50902294ce9a  15:31:18                                    5.73  5.73  5.73  81
770004242c2d  15:31:30                                    5.48  5.49  5.48  121
688268302c2d  15:31:58  706.734 699.743 18   0     0                              136
51705369ce9a  15:31:58  681.166 677.847 17   6     0                              94
51705516ce9a  15:32:20  682.337 677.847 17   0     0                              84
770004242c2d  15:29:54                                          5.48  5.49  5.48  159
51705369ce9a  15:30:21  681.16  677.847 17   0     0                              104
688268302c2d  15:30:22  706.734 699.743 18   0     0                              154
51705516ce9a  15:30:43  682.337 677.847 17   0     0                              126
50902542ce9a  15:30:49                                    5.59  5.62  5.6   128
50902294ce9a  15:31:17                                    5.73  5.73  5.73  143
770004242c2d  15:31:28                                    5.48  5.49  5.48  159
688268302c2d  15:31:57  706.734 699.743 18   0     0                              153
51705369ce9a  15:31:57  681.166 677.847 17   6     0                              104
51705516ce9a  15:32:19  682.337 677.847 17   0     0                              126
```

Figure 6.2: Formatted wM-Bus packets showing which meter that sent it, the time it was received, current volume registered, volume at beginning of the month, ambient temperature, volume difference from last read, volume difference from last read of monthly volume, minimum pressure registered, maximum pressure registered, instant pressure registered and lastly the received signal strength (RSSI). Top figure are packets received by mbus-collector-1, while bottom are packets received by mbus-collector-2.

sending out packets, the toggling light, as defined in the code, was checked. This worked as expected. In a similar manner as the real meters, the raw wM-Bus packets were inspected in the saved local file. A comparison was made with the data used to load onto the meter's flash storage. Since they were identical, this confirmed that the simulated meter code and the decryption in the collector worked as expected.

With communication between meters and collector working correctly, two collectors were then used further and all simulated meters were placed at various locations to test signal strength. mbus-collector-2 was located next to mbus-collector-1 at an office. Because of restrictions due to an ongoing pandemic, the application Teamviewer was used extensively to interact with the two computers used in the gateway. This way, changes to the code and configuration could be made remotely.

Packet formatting was then tested. The serial port logging code formatted the received wM-Bus packets and printed them to the console window, as well as saved them locally to a file. Figure 6.2 shows the printed results from both collectors running on each their own computer. To make sure the values made sense, they were graphed with a simple script, as shown in figure 6.3. The graph shows that pressure values changes as there is a consumption of water. A value range between 5 and 6 bars for pressure and a couple of litres for consumption are believable values, and it is concluded that the formatting of packets worked as expected.

6.2 Testing Cloud Solution Components

With correct values being read from the WDN, the first two layers were working as expected. The data could then be uploaded to the cloud solution on the Device & Data Management Layer.

6.2.1 Receiving data in IoT Core

The last responsibility of the serial port logging code was then to transfer the data to IoT Core. To test this out, the IoT Core provides an MQTT test client to see if data is received on a specific MQTT topic. The topic filter '#' was used to subscribe to any topic being received. The result

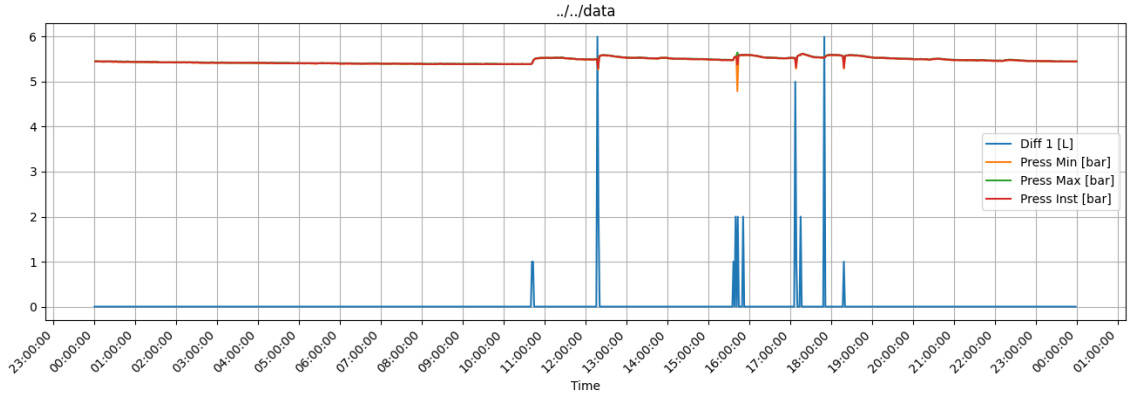


Figure 6.3: Formatted wM-Bus packets graphed to visualize the values that was outputted by the serial port logging code. Diff 1 is the difference in volume for two readings.

is shown in figure 6.4. The data was received by both collectors in a JSON format, as specified by the serial port logging code.

6.2.2 Test Query of Timestream

Next step was to see if the IoT rule worked properly by transfering the JSON packets to the Timestream database. Using the Query Editor for Timestream, the database can be queried for its content. A simple SQL query was made:

```
SELECT *
FROM "sensor_data"."collector_data"
WHERE time between ago(15m) and now()
ORDER BY time DESC LIMIT 50
```

The result of the query is shown in figure 6.5. Several pages worth of data were stored in the 15 minute time range. This confirmed that data was being stored in the database and the third layer therefore worked.

6.2.3 Test of GET Method for API Gateway

To test the next layer containing the lambda function and the API Gateway, it had to get a request from an end user in the last layer. An application called Postman is used for this purpose. Postman is a specialized application for API development and can be used to send and receive API requests. Using this application, the following GET request was made:

```
https://doqhob3z60.execute-api.eu-west-1.amazonaws.com/test/?measure_name=inst_pressure&Location=loc-1
```

This GET request asks the API gateway to return packets containing inst_pressure at loc-1. The result is shown in figure 6.6. Since this only served as a demonstration, the resulting API response is far from optimized as it sends all data as is, but this shows that all layers of the infrastructure worked as expected. From a meter broadcasting its reading, to a third-party application reading the value.

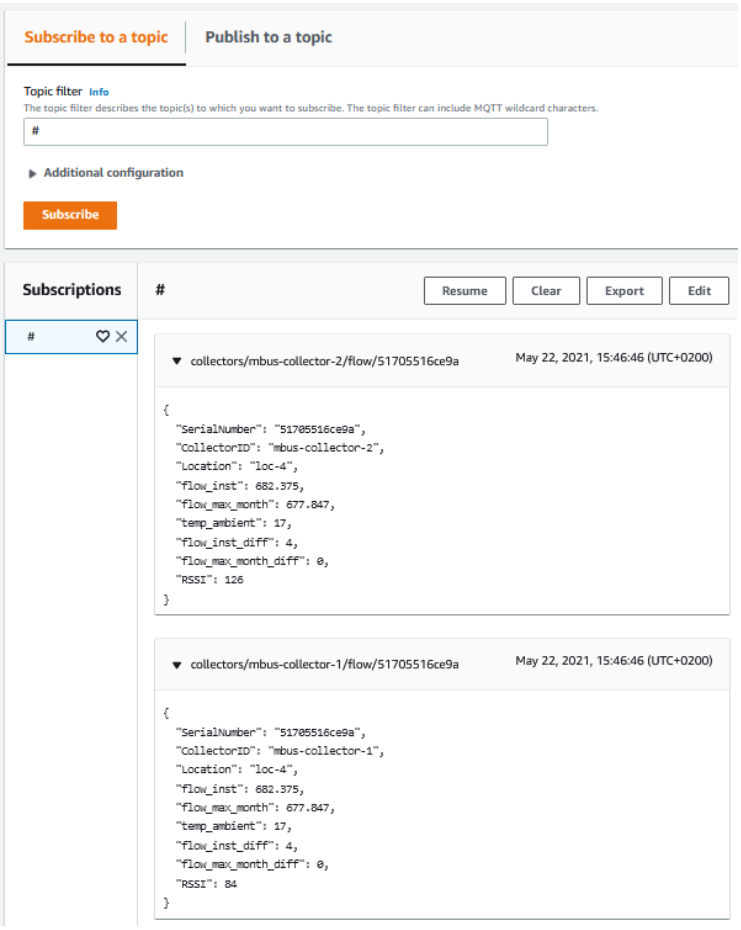
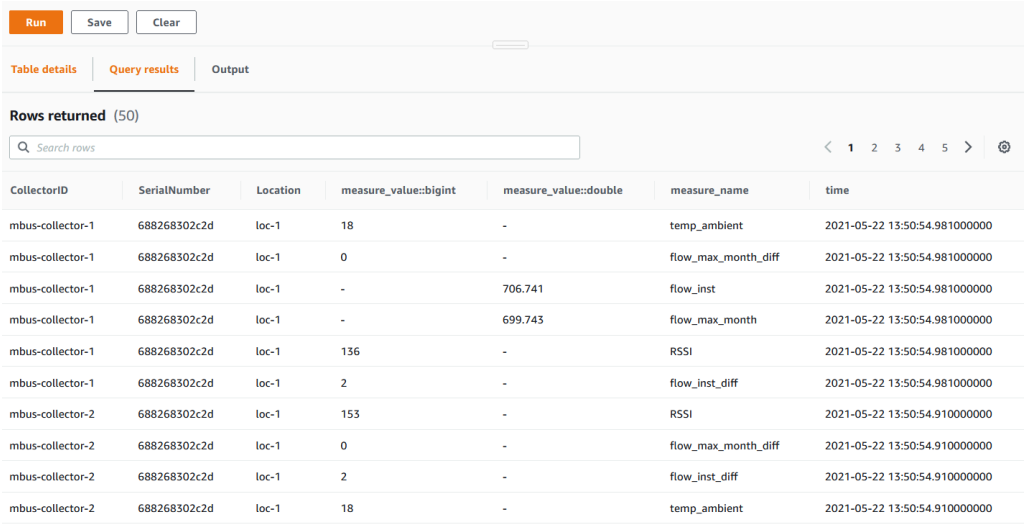
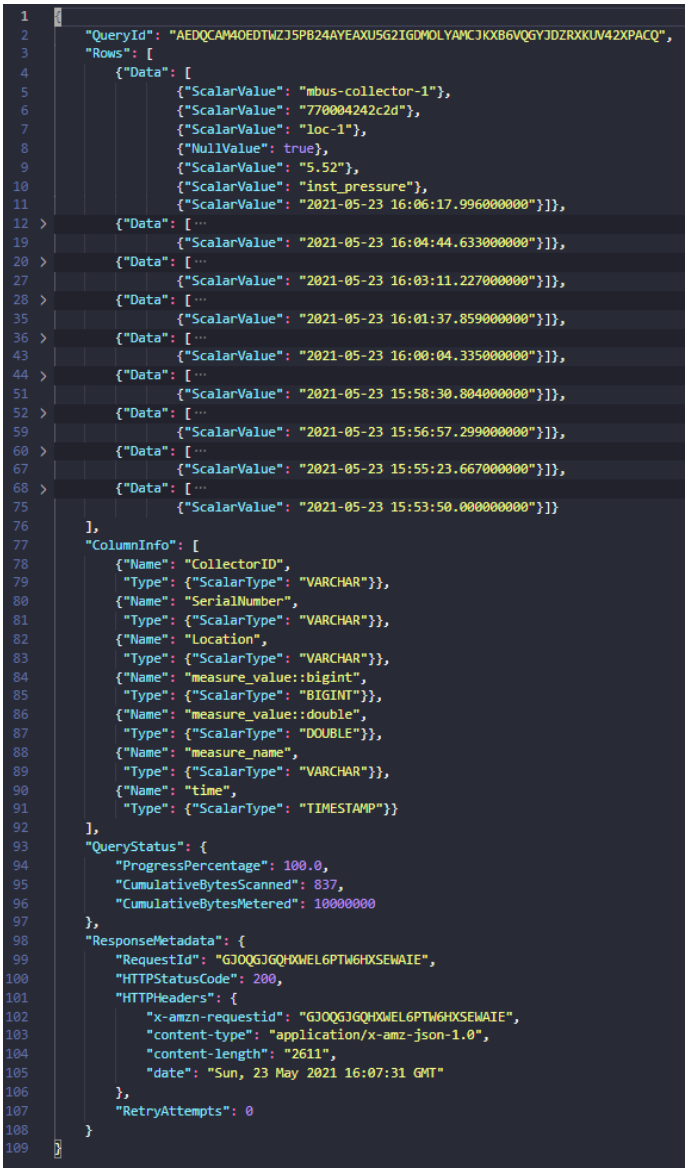


Figure 6.4: JSON packets being received by the IoT Core on a specific filter as shown. A new timestamp is added to the packets.



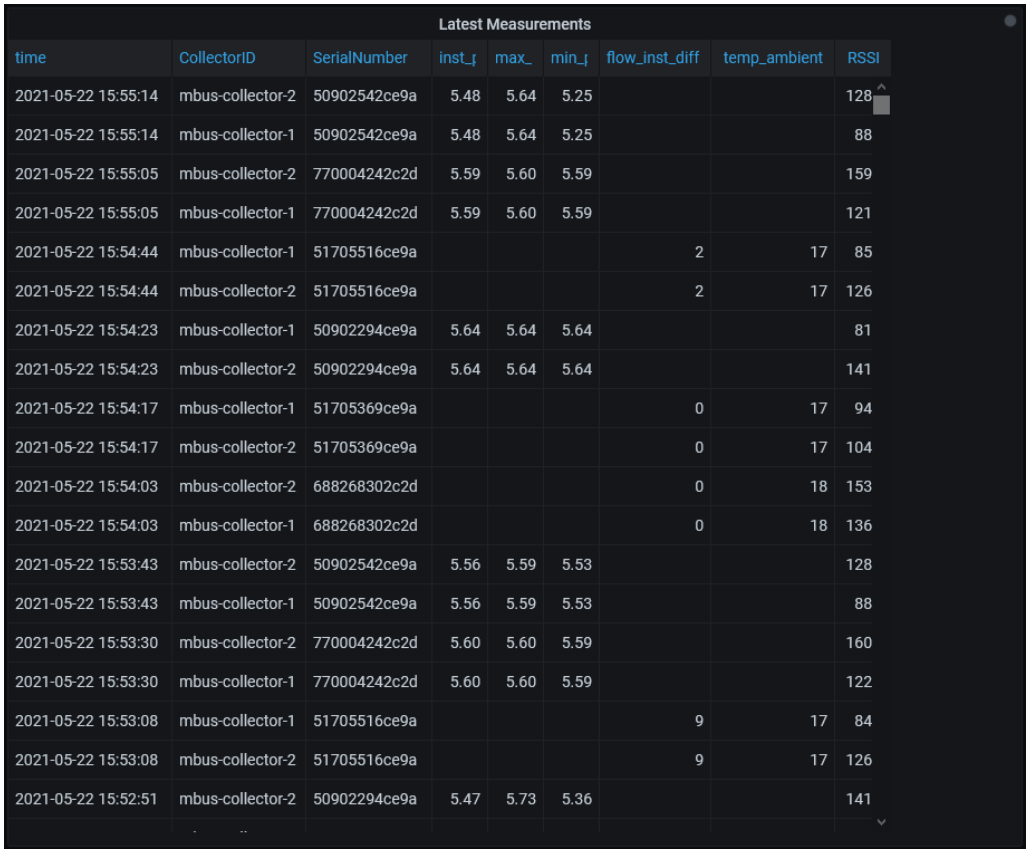
CollectorID	SerialNumber	Location	measure_value::bigint	measure_value::double	measure_name	time
mbus-collector-1	688268302c2d	loc-1	18	-	temp_ambient	2021-05-22 13:50:54.981000000
mbus-collector-1	688268302c2d	loc-1	0	-	flow_max_month_diff	2021-05-22 13:50:54.981000000
mbus-collector-1	688268302c2d	loc-1	-	706.741	flow_inst	2021-05-22 13:50:54.981000000
mbus-collector-1	688268302c2d	loc-1	-	699.743	flow_max_month	2021-05-22 13:50:54.981000000
mbus-collector-1	688268302c2d	loc-1	136	-	RSSI	2021-05-22 13:50:54.981000000
mbus-collector-1	688268302c2d	loc-1	2	-	flow_inst_diff	2021-05-22 13:50:54.981000000
mbus-collector-2	688268302c2d	loc-1	153	-	RSSI	2021-05-22 13:50:54.910000000
mbus-collector-2	688268302c2d	loc-1	0	-	flow_max_month_diff	2021-05-22 13:50:54.910000000
mbus-collector-2	688268302c2d	loc-1	2	-	flow_inst_diff	2021-05-22 13:50:54.910000000
mbus-collector-2	688268302c2d	loc-1	18	-	temp_ambient	2021-05-22 13:50:54.910000000

Figure 6.5: Result of query for Timestream. Several pages worth of data is shown for a 15 minute data extraction. The data structure is changed when it is stored to Timestream.



```
{
    "QueryId": "AEDQCAM4OEDTWZJ5PB24AYEAXU5G2IGDMOLYAMCJKXB6VQGYJDZRXKUV42XPACQ",
    "Rows": [
        {"Data": [
                {"ScalarValue": "mbus-collector-1"},
                {"ScalarValue": "770004242c2d"},
                {"ScalarValue": "loc-1"},
                {"NullValue": true},
                {"ScalarValue": "5.52"},
                {"ScalarValue": "inst_pressure"},
                {"ScalarValue": "2021-05-23 16:06:17.996000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 16:04:44.633000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 16:03:11.227000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 16:01:37.859000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 16:00:04.335000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 15:58:30.804000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 15:56:57.299000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 15:55:23.667000000"}]},
        {"Data": [···
                {"ScalarValue": "2021-05-23 15:53:50.000000000"}]}
    ],
    "ColumnInfo": [
        {"Name": "CollectorID",
         "Type": {"ScalarType": "VARCHAR"}},
        {"Name": "SerialNumber",
         "Type": {"ScalarType": "VARCHAR"}},
        {"Name": "Location",
         "Type": {"ScalarType": "VARCHAR"}},
        {"Name": "measure_value::bigint",
         "Type": {"ScalarType": "BIGINT"}},
        {"Name": "measure_value::double",
         "Type": {"ScalarType": "DOUBLE"}},
        {"Name": "measure_name",
         "Type": {"ScalarType": "VARCHAR"}},
        {"Name": "time",
         "Type": {"ScalarType": "TIMESTAMP"}}
    ],
    "QueryStatus": {
        "ProgressPercentage": 100.0,
        "CumulativeBytesScanned": 837,
        "CumulativeBytesMetered": 10000000
    },
    "ResponseMetadata": {
        "RequestId": "GJOQGJGQHXWEL6PTW6HXSEWAIE",
        "HTTPStatusCode": 200,
        "HTTPHeaders": {
            "x-amzn-requestid": "GJOQGJGQHXWEL6PTW6HXSEWAIE",
            "content-type": "application/x-amz-json-1.0",
            "content-length": "2611",
            "date": "Sun, 23 May 2021 16:07:31 GMT"
        },
        "RetryAttempts": 0
    }
}
```


Figure 6.6: Result of Get API response. All data from the Timestream query was saved and returned by Lambda.



Latest Measurements

time	CollectorID	SerialNumber	inst_f	max_	min_f	flow_inst_diff	temp_ambient	RSSI
2021-05-22 15:55:14	mbus-collector-2	50902542ce9a	5.48	5.64	5.25			128
2021-05-22 15:55:14	mbus-collector-1	50902542ce9a	5.48	5.64	5.25			88
2021-05-22 15:55:05	mbus-collector-2	770004242c2d	5.59	5.60	5.59			159
2021-05-22 15:55:05	mbus-collector-1	770004242c2d	5.59	5.60	5.59			121
2021-05-22 15:54:44	mbus-collector-1	51705516ce9a				2	17	85
2021-05-22 15:54:44	mbus-collector-2	51705516ce9a				2	17	126
2021-05-22 15:54:23	mbus-collector-1	50902294ce9a	5.64	5.64	5.64			81
2021-05-22 15:54:23	mbus-collector-2	50902294ce9a	5.64	5.64	5.64			141
2021-05-22 15:54:17	mbus-collector-1	51705369ce9a				0	17	94
2021-05-22 15:54:17	mbus-collector-2	51705369ce9a				0	17	104
2021-05-22 15:54:03	mbus-collector-2	688268302c2d				0	18	153
2021-05-22 15:54:03	mbus-collector-1	688268302c2d				0	18	136
2021-05-22 15:53:43	mbus-collector-2	50902542ce9a	5.56	5.59	5.53			128
2021-05-22 15:53:43	mbus-collector-1	50902542ce9a	5.56	5.59	5.53			88
2021-05-22 15:53:30	mbus-collector-2	770004242c2d	5.60	5.60	5.59			160
2021-05-22 15:53:30	mbus-collector-1	770004242c2d	5.60	5.60	5.59			122
2021-05-22 15:53:08	mbus-collector-1	51705516ce9a				9	17	84
2021-05-22 15:53:08	mbus-collector-2	51705516ce9a				9	17	126
2021-05-22 15:52:51	mbus-collector-2	50902294ce9a	5.47	5.73	5.36			141

Figure 6.7: Grafana panel showing the latest packets that have been registered.

6.3 Visualization of Data using Grafana

To visualize the data contained in the Timestream database, Grafana was used. By connecting directly to the database through a Grafana plugin, it was able to query the Timestream database directly. to verify packets being received, a table of the last packets was created, as shown in figure 6.7. A graph for each location of meters were created, as shown in figure 6.8. Here, only location 1 contains real meters, while other locations contain data from simulated meters. Lastly, a graph for both collectors was created to show the signal strength of the received packets from all meters, as shown in figure 6.9.

6.4 Generated Pressure Residuals

The real instant pressure and the modelled pressure from EPANET was plotted on the same graph along with the estimated consumption flow for an arbitrary day to see how they differed. The result is shown in figure 6.10. The difference, also called the pressure residual, was plotted on its own graph, as shown in figure 6.11. An initial pressure at the inlet was chosen to be 5.96 bar. This is the average pressure for the time 00:00 to 03:00, which is expected to have minimum consumption of water. The estimated water demand pattern was calibrated to give a similar response on the modelled pressure as the real one. Several water demand patterns had to be generated using the script in appendix D, and then tested on EPANET. After several trials, a multiplier of 120, as shown on code line 46 in appendix D, was shown to match the real pressures the best.

Figure 6.8: Grafana panel showing data for all locations. Location 1 contains the real Kamstrup meters, while the others are simulated.

Figure 6.9: Grafana panel showing RSSI measurements from all meters for both collectors.

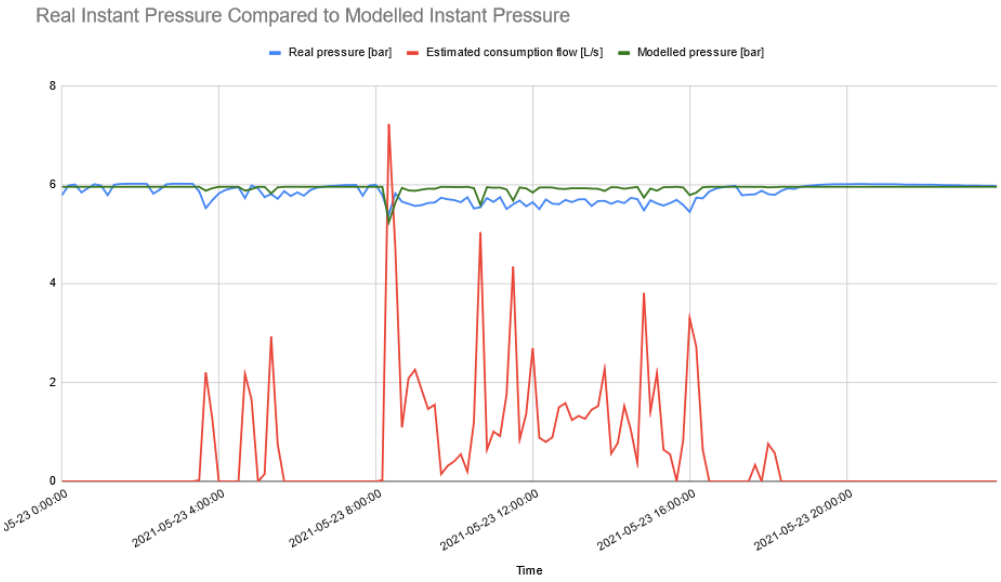


Figure 6.10: Real pressure measurements plotted alongside modelled pressure estimations. Consumption flow is averaged from measured volume consumed in a time range, and is also multiplied by 120.

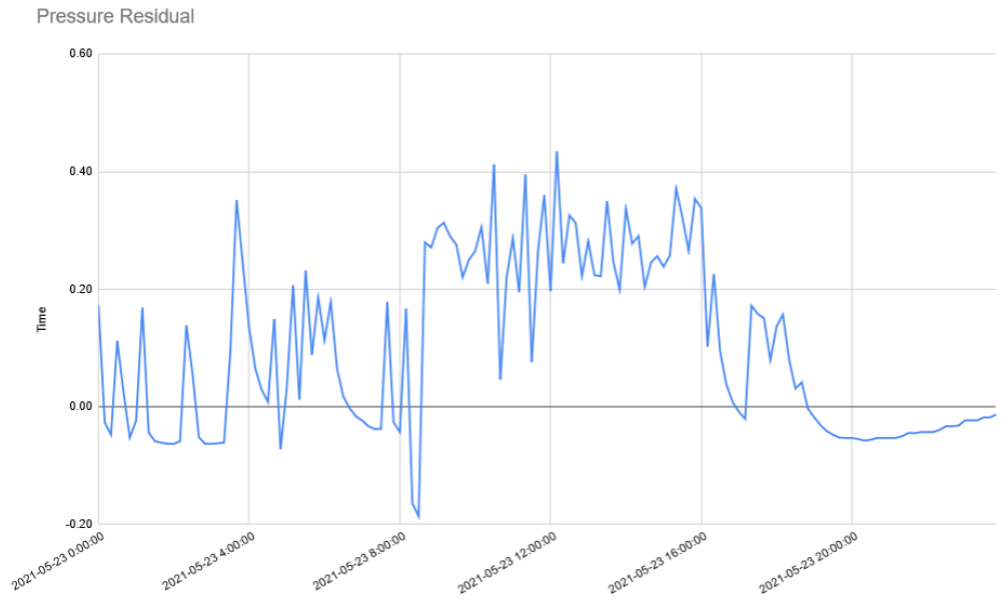


Figure 6.11: The pressure residual generated from a difference in real pressure value and modelled pressure estimation.

Chapter 7

Discussion

The purpose of the project was to develop an infrastructure for Collecting and Analysing near Real-Time Data from Several Water Meters Using Wireless M-Bus. By using a layered approach, as suggested from the literature, the various parts were implemented step-wise. A promising method of using pressure residuals from a model was investigated.

7.1 Review of Specifications

The collector was able to receive wM-Bus packets using C1 mode in a unidirectional manner, therefore fulfilling **specification one** in chapter 4. Packets from both flow meter and pressure meter by Kamstrup were received by the collector, and **specification two** was fulfilled. The implemented simulated meters were able to send its content of either pressure data or flow to the collector, which then fulfills **specification three**. Since the meters needed to be installed on the collectors, this fulfills **specification four** of only receiving wM-Bus packets from specified meters. A key had to be provided to make the packets content readable, therefore fulfilling **specification five**. A gateway was implemented and worked successfully by transmitting meter data to the Timestream database, which fulfilled **specification six**. Since the gateway formatted the wM-Bus packets before they were stored, this fulfilled **specification seven**.

Specification eight required the data to be easily accessible at a later time, this is fulfilled to some degree. As shown, the data is available through API requests made by API Gateway. Both long-term magnetic storage and short-term in-memory storage is available through API requests. The API response processing is too simple to be a complete solution, and more work needs to be done on it. Only a GET response was implemented, which gave a non-optimal response, as shown in figure 6.6. Grafana was used to visualize the data. This worked well for the few meters that the infrastructure contained, but it was noticeably slow after all graphs had been added. Each graph performed a query similar to the one shown in code listing 5.4, and it is suspected that it is a inefficient way to do it. The query looked like this since the flat structure of the time series data needed to be converted into one with several values for each entry, as shown in table 5.5. Another solution would be to process the data and store it in another type of structure in the cloud. This was not possible using the Timestream service, which automatically used a flat structure for the data records.

Specification nine concerning range requirement is also somewhat fulfilled. The two collectors are placed in the same office, where mbus-collector-2 uses an external dipole antenna, while mbus-collector-1 uses a smaller whip antenna. The external dipole antenna shows to improve the measured RSSI value for the signals, as shown in figure 6.9. The flow meter 688268302c2d and simulated meter 51705369 seems to have the least improvement in signal strength. Pressure meter 770004242 seems to have an erratic, but better signal strength. it is difficult to verify that the specification holds and more investigation needs to be conducted. From the literary study, it is safe to assume there are some concerns with the range capabilities, as table 2.1 shows wM-Bus to have the lowest range capability of 500m for 868MHz in line of sight. The problem can be solved with several gateways, but this could lead to higher costs. A lightweight and cheaper gateway alternative could be developed for this purpose and make the solution viable. Another solution would be to use a wM-Bus mode supporting lower frequencies. Since the data rate would be lower, this could hurt the viability of leakage detection and localization strategies. The conclusion for specification nine is that more work needs to be done.

Specification ten of making analysis of leakages possible is fulfilled to some degree. The result of generating pressure residuals was shown to be inaccurate as expected, but the proof-of-concept still holds. Several assumptions made the results not viable. A complete description of a DMA was not available, and only a snapshot of nearby pipe grid was used in the construction of the model. Accurate coefficients for the pipes were also not available. Therefore, a default coefficient for cast-iron with a value of 100 was used. All nodes were assumed to by at a height of 0. A pipe node $J1$ was assumed to be a reservoir and an assumed head based on minimum night flow during 00:00 to 03:00 for node $J5$ was used as base pressure. The inlet of a DMA is also controlled by a pressure reduction valve, and the behaviour of this valve was not known. Node $J3, J4, J5, J7$ and $J8$ were also assumed to have a similar demand pattern as node $J5$, the location of the real meters. For future work, more data of the DMA needs to be acquired to make the model viable. The accuracy of demand data should also be improved by introducing more flow meters to the system. By also introducing more pressure meters, several pressure residual data sets can be generated to be used further for various leakage detection and localization techniques.

Some things can be said about the available results of the pressure residual. From the start of the time range, there are pressure variations which are not accounted by the water consumption of the meter. This could be variations made by the pressure reduction valve at the inlet of the DMA, or it could be an unknown water consumption. It is also shown that the model overcompensates at certain water consumption peaks, such as at 08:00, and not on others, such as at 04:00. This could be because of other unknown water consumption, or that the time range of tapping differs considerably. When there is a steady consumption of water, as for 08:00 to about 16:00, the pressure is at a lower level than estimated by the model. In total, it is suspected that big contributions to these observations are changes made to the pressure reduction valve, other unknown consumption of water and the time range of the tapping. If a leak occurs, a new constant flow rate will occur. This is difficult to detect if the pressure reduction valve adjusts itself to compensate. It is also difficult to know when every node's water consumption is not known.

The way the system works now, the data needs to be extracted from the infrastructure in order to perform offline data analysis. It would be beneficial to have the model as a part of the online system with pressure residuals being calculated automatically. As the infrastructure on figure

4.1 shows, the system is quite modular. With specific tasks being tied to services, extra services can be added to implement more functionality to the cloud solution.

Specification eleven of managing all meters in a timely manner is fulfilled for this use case, but when it is scaled up with more meters, it is uncertain. The cloud solution is said to be serverless, so increased demands cause automatically scaling which requires no intervening. A weakness of the infrastructure is that the serial port logger code adds a timestamp to wM-Bus packets when received and IoT core adds another timestamp for when the cloud receives the packet. This means that the packets have a delay from reading time to when the cloud receives it. The wM-Bus packets has a timestamp which is sent alongside the meter data, but the problem is that it is relative. A future solution could then implement a gateway which keep a record of all relative times in order to improve time syncronization. As figure 6.3 shows, there is a time delay of two seconds for some records on the different collectors. Figure 6.4 though, shows a record that has the same time on the different collectors. This shows that the current infrastructure might have a time uncertainty somewhere around a second, which could be improved by using the relative time offered by the wM-Bus packets. For the Kamstrup packets, this is a part of session number (SN), as shown in figure A.2. For the simulated meters, it is a part of the application layer, as shown in figure A.1. The use of office computers running a script is also not sufficient in a real system of a bigger scale. As mentioned, a more lightweight solution for gateway would be required. To make this work, pre-processing of data might have to be moved over to the cloud solution as the gateway might not be powerful enough to handle all meters in a timely manner.

7.2 Other Improvements

As this project only investigated AWS as a possible central storage solution, there might be others more suitable. Costs and privacy might be a concern which should be looked into, and a privately hosted solution might solve these issues. This system costed about 49.53 dollars to be hosted for about five months. It was experienced that some services on AWS brought higher costs with it than others. It is therefore suspected that a lot of cost optimizations can be made based on certain choices. Since the system is modular, it is possible to try out different implementations for the various layers, as long as the interface between them holds.

As other LPWAN technologies have been shown to be promising in use for water meters, there might arise a situation where many different communication technologies are in use for a system. It would then be beneficial to implement a gateway able to handle several protocols. The gateway could then send it to a common storage solution.

Chapter 8

Conclusion

In this project, an infrastructure for collecting pressure and volume measurements in near real-time for several households in a neighbourhood was created. The data collected was then used to generate pressure residuals necessary for several model-driven leakage detection and localization methods. It was shown how introducing pressure meters in addition to volume meters, as desired by many municipals in Norway, it is possible to create a system for leakage localization inside a District Metered Area (DMA). A caveat of this approach was shown to be the extensive knowledge required of a DMA to create a model sufficient for generating viable pressure residuals. Although the system is working as is, several aspect may be improved in future works.

Chapter 9

Further Work

A complete infrastructure for data collection and analysis has many areas of improvement that can be looked into in further works. This chapter will lay out things mentioned in the discussion in chapter 7. There are two main areas of improvement; either focused on infrastructure or focused on analysis.

9.1 Further Work for Infrastructure

A major improvement would be to increase the number of real meters available for the infrastructure as it would make the leakage analysis more robust. More volume meters would make the demand estimation more accurate and more pressure meters would make it possible to generate more pressure residuals necessary for further analysis.

it is worth looking into if other LPWAN protocols offering longer range can be used for a model-driven leakage analysis. This could increase the range capabilities limiting wM-Bus, but data rates might be a concern. Also mode N for wM-Bus offers a longer range and might be viable, but it is not supported by the OMS standard yet.

The gateway should be made more lightweight so that it doesn't require a development kit connected to a personal computer. It could be run on a smaller device instead to reduce costs or be custom-made. This would also make it more viable to have a higher density of gateways in a neighbourhood, making range concerns less problematic.

It might be more beneficial to perform formatting of wM-Bus packets in the cloud rather than in the gateway layer, since the gateway might not have those capabilities when scaling up the number of smart meters and making it more lightweight.

Better time synchronization of wM-Bus packets should be done. This project used a timestamp given by the cloud for packets. There is a relative time associated with every packet that can be used instead. This would improve delays associated with uploading the packets. The infrastructure would then have to keep track of relative times of all meters. Maybe, with two-way communication, it is possible to synchronize all meters to a common time defined by the collector.

A more thorough API could be made to make meter data more available for third party applic-

ations. Data could then be downloaded directly from the cloud solution.

Security, privacy and cost considerations have not been taken into account. There might be benefits of using another type of central storage architecture than a cloud platform hosted by a private company. Also, a thorough comparison of other available cloud platforms can be looked into.

9.2 Further Work for Data Analysis

The implementation of a online model is desirable so that a model-based leakage analysis doesn't require off-line work, and therefore make it more automated.

More information about the WDN needs to be attained for a thorough model to be made. A model-driven approach' accuracy is hugely dependent on how accurate the model is. This means getting information about pipe length, pipe diameter, roughness coefficients, flow and pressure data from entrance of a DMA.

With a robust way of generating pressure residuals in place, an investigation into viable model-driven approaches need to be made. As concluded in the literary study, a hybrid method of using residuals for a classifier seems to be a promising solution.

Bibliography

[1] C. G. Skjæveland, 'Establishing communication with wireless m-bus using equipment from different vendors,' 2020.

[2] D. Zaman, M. K. Tiwari, A. K. Gupta and Dhrubjyoti, 'A review of leakage detection strategies for pressurised pipeline in steady-state,' 2019.

[3] H. H. Lier, 'Demonstrering av konsept for innsamling og sammenstilling av data fra flere vannmålere ved bruk av trådløs m-bus,' 2018.

[4] A. Vatland, 'Wireless m-bus communication between equipment from different vendors,' 2020.

[5] A. Soldevila, J. Blesa, S. Tornil-Sin, E. Duviella, R. M. Fernandez-Canti and V. Puig, 'Leak localization in water distribution networks using a mixed model-based/data-driven approach,' 2016.

[6] Y. Lalle, M. Fourati, L. C. Fourati and J. P. Barraca, 'Communication technologies for smart water grid applications: Overview, opportunities, and research directions,' 2021.

[7] A. Pitì, G. Verticale, C. Rottondi, A. Capone and L. L. Schiavo, 'The role of smart meters in enabling real-time energy services for households: The italian case,' 2017.

[8] G. Antzoulatos, C. Mourtzios, I.-O. K. Panagiota Stournara, N. Papadimitriou, D. Spyrou, A. Mentes, E. Nikolaidis, A. Karakostas, D. Kourtesis, S. Vrochidis and I. Kompatsiaris, 'Making urban water smart: The smart-water solution,' 2020.

[9] W. Anani, A. Ouda and A. Hamou, 'A survey of wireless communications for iot echosystems,' 2019.

[10] P. Ferrari, A. Flammini, S. Rinaldi and A. Vezzoli, 'Wireless sensor network based on wm-bus for leakage detection in gas and water pipes,' 2013.

[11] J. Fiedler, 'An overview of pipeline leak detection technologies,' 2014.

[12] M. V. Casillas, L. E. Garza-Castañón and V. Puig, 'Model-based leak detection and location in water distribution networks considering an extended-horizon analysis of pressure sensitivities,' 2014.

[13] L. Magenta, 'Energy potentiality assessment by mean of intelligent pressure management in the hydraulic network of trondheim,' 2018.

[14] 'Open Metering System Specification - Primary Communication,' OMS Group, Cologne, DE, Standard, Nov. 2019.

[15] 'Communication systems for meters Part 7: Transport and security services,' European Committee for Standardization, Brussels, BE, Standard, Jul. 2018.

[16] 'Communication systems for meters Part 4: Wireless M-Bus communication,' European Committee for Standardization, Brussels, BE, Standard, May 2019.

[17] *Wireless M-Bus Documentation: Quick Start Guide*, RCxxxxDK-USB, STACKFORCE, Mar. 2015.

[18] STACKFORCE. (2020). 'Wireless M-Bus Stack,' [Online]. Available: https://www.stackforce.de/en/products/protocol-stacks/wireless-m-bus-stack (visited on 16/12/2020).
[19] Aprova AS. (2021). 'epanet.no,' [Online]. Available: https://epanet.no/kom-i-gang/teori/hydraulikk-i-epanet/ (visited on 23/04/2021).
[20] U. S. E. P. Agency. (2021). 'EPANET documentation,' [Online]. Available: https://epanet22.readthedocs.io/en/latest/ (visited on 24/04/2021).
[21] E. Todini and L. A. Rossman, 'Unified framework for deriving simultaneous equation algorithms for water distribution networks,' 2013.
[22] R. Ben-Mansour, M. Habib, A. Khalifa, K. Youcef-Toumi and D.Chatzigeorgiou, 'Computational fluid dynamic simulation of small leaks in water pipelines for direct leak pressure transduction,' 2012.
[23] A. Abdulshaheed, F. Mustapha and A. Ghavamian, 'A pressure-based method for monitoring leaks in a pipe distribution system: A review,' 2017.
[24] S. Labs. (2020). 'Wireless M-Bus Software,' [Online]. Available: https://www.silabs.com/development-tools/wireless/proprietary/ezr32wg-868-mhz-starter-kit (visited on 09/12/2020).
[25] *RCxxxxDK-USB Demonstration Kit User Manual*, RCxxxxDK-USB, Radiocrafts, 2018.
[26] Radiocrafts. (2018). 'RCTools MBUS,' [Online]. Available: https://radiocrafts.com/resources/supporting-software-and-software-tools/ (visited on 09/12/2020).
[27] Amazon Web Services, Inc. or its affiliates. (2021). 'Amazon Timestream Architecture,' [Online]. Available: https://docs.aws.amazon.com/timestream/latest/developerguide/architecture.html (visited on 20/05/2021).
[28] Amazon Web Services, Inc. or its affiliates. (2021). 'Amazon Timestream Concepts,' [Online]. Available: https://docs.aws.amazon.com/timestream/latest/developerguide/concepts.html (visited on 20/05/2021).
[29] M. Fagiani, S. Squartini, L. Gabrielli, M. Severini and F. Piazza, 'A statistical framework for automatic leakage detection in smart water and gas grids,' 2016.
[30] P. Masek, D. Hudec, J. Krejci, A. Ometov, J. Hosek, S. Andreev, F. Kropfl and Y. Koucheryavy, 'Advanced wireless m-bus platform for intensive field testing in industry 4.0-based systems,' 2018.
[31] A. Sikora, P. Lehmann, N. Anantalapochai, M. Dold, D. Rahusen and A. Rohleder, 'Recent advances in en13757 based smart grid communication,' 2014.
[32] S. Alshattnawi, 'Smart water distribution management system architecture based on internet of things and cloud computing,' 2017.
[33] J. Guth, U. Breitenbücher, M. Falkenthal, P. Fremantle, O. Kopp, F. Leymann and L. Reinfurt, 'A detailed analysis of iot platform architectures: Concepts, similarities, and differences,' 2018.
[34] *UG200: EZR32WG 868 MHz 13 dBm Wireless Starter Kit User's Guide*, SLWSTK6220A, Rev. 2.01, Silicon Labs, Dec. 2017.
[35] *Datasheet: Kamstrup PressureSensor*, 58101424, Kamstrup, Aug. 2017.
[36] Kamstrup. (2021). 'Communication Technologies for Water Meter Reading,' [Online]. Available: https://www.kamstrup.com/en-en/water-solutions/water-meter-reading/communication-technologies (visited on 28/04/2021).
[37] 'Communication systems for meters Part 3:Application protocols,' European Committee for Standardization, Brussels, BE, Standard, Jul. 2018.
[38] 'Open Metering System Specification - General Part,' OMS Group, Cologne, DE, Standard, Oct. 2014.

Appendix A

M-Bus Packets

EZR32WG pressure packet

Not encrypted																
Data Link Layer										Transport Layer Header						
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	ACC	STS	CF	CF	DV	DV
2f	44	9a	ce	1	0	0	80	1	18	7a	21	0	20	25	2f	2f

Encrypted																										
Application Layer with full M-Bus frame																										
DIF	VIF	ss	mm	hh	DD	MM	YY	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22
6	6d	9a	13	13	2	1	0	64	0	65	0	64	0	bd	cf	a4	3a	e7	36	85	3a	8	0	2f	2f	2f	2f	2f	2f

EZR32WG flow packet

Not encrypted																
Data Link Layer										Transport Layer Header						
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	ACC	STS	CF	CF	DV	DV
2e	44	9a	ce	1	0	0	80	1d	16	7a	21	0	20	25	2f	2f

Encrypted																										
Application Layer with full M-Bus frame																										
DIF	VIF	ss	mm	hh	DD	MM	YY	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22
6	6d	9a	13	13	2	1	0	0	0	39	22	0a	0	0f	0b	0a	0	13	2f	2f	2f	2f	2f	2f	2f	2f	2f	2f	2f

Figure A.1: SLWSTK6220A wM-Bus packets used in this project. Yellow highlighted fields indicate the measurement data that was extracted from Kamstrup meters. For the pressure packet, first yellow group is minimum pressure, second is maximum pressure, third is instant pressure, the two next are manufacturer specific, and the last is error flags. For the flow packet, first yellow group is info codes, second is instant flow, third is monthly flow, and last is ambient temperature. No data information blocks (DIB) or value information blocks (VIB) are sent. Example values are shown.

Kamstrup flowIQ 3100

Not encrypted																		
Data Link Layer										Extended Link Layer								
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	CC	ACC	SN1	SN2	SN3	SN4	CRC	CRC
22	44	2d	2c	30	68	82	68	1d	16	8d	20	87	c0	4	8e	21	52	5c

Encrypted															
Application Layer with compact M-Bus frame															
CI	FS	FS	FCRC	FCRC	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11
79	4	3f	70	47	0	0	39	22	0a	0	0f	0b	0a	0	13

Kamstrup flowIQ 3100

Not encrypted																		
Data Link Layer										Extended Link Layer								
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	CC	ACC	SN1	SN2	SN3	SN4	CRC	CRC
27	44	2d	2c	30	68	82	68	1d	16	8d	20	88	d4	4	8e	21	67	e8

Encrypted																		
Application Layer with full M-Bus frame																		
CI	DIF	VIF	VIFE	D1	D2	DIF	VIF	D3	D4	D5	D6	DIF	VIF	D7	D8	D9	D10	DIF	VIF	D11
78	2	ff	20	0	0	4	13	39	22	0a	0	44	13	0f	0b	0a	0	61	67	13

Kamstrup PressureSensor

Not encrypted																		
Data Link Layer										Extended Link Layer								
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	CC	ACC	SN1	SN2	SN3	SN4	CRC	CRC
27	44	2d	2c	24	4	0	77	1	18	8d	28	2b	4c	91	89	21	ac	df

Encrypted																				
Application Layer with compact M-Bus frame																				
CI	FS	FS	FCRC	FCRC	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16
79	de	73	de	1e	64	0	65	0	64	0	bd	cf	a4	3a	e7	36	85	3a	8	0

Kamstrup PressureSensor

Not encrypted																		
Data Link Layer										Extended Link Layer								
L	C	M1	M2	A1	A2	A3	A4	V	T	CI	CC	ACC	SN1	SN2	SN3	SN4	CRC	CRC
32	44	2d	2c	24	4	0	77	1	18	8d	28	f1	c4	f1	88	21	10	bf

Encrypted																														
Application Layer with full M-Bus frame																														
CI	DIF	VIF	D1	D2	DIF	VIF	D3	D4	DIF	VIF	D5	D6	DIF	VIF	VIFE	D7	D8	D9	D10	DIF	VIF	VIFE	D11	D12	D13	D14	DIF	VIF	VIFE	D15	D16
78	22	69	66	0	12	69	66	0	2	69	66	0	5	ff	9	b4	21	b0	3a	5	ff	0a	4b	1c	bb	3a	2	fd	17	8	0

Figure A.2: Kamstrup M-Bus packets used in this project. Yellow highlighted fields indicate the measurement data. For the pressure packet, first yellow group is minimum pressure, second is maximum pressure, third is instant pressure, the two next are manufacturer specific, and the last is error flags. For the flow packet, first yellow group is info codes, second is instant flow, third is monthly flow, and last is ambient temperature. Example values are shown.

Appendix B

Serial Port Logging Code

```
"""
Required packages:
python3
Microsoft C++ Build Tools (Microsoft Visual C++ 14.0)
pip install requests pyserial python_jwt sseclient pycryptodome requests-toolbelt AWSIoTPythonSDK
* Might have to change crypto to Crypto in AppData\Local\Programs\Python\Python39\Lib\site-packages
"""

from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
import serial
import serial.tools.list_ports as list_ports
import csv
from datetime import datetime
import time
import json

################################################################################
# Global variables
################################################################################

# Constants for AWS cloud upload
clientId = "mbus-collector-2"
host = "a2ap02hejjfikb-ats.iot.eu-west-1.amazonaws.com"
cloud_port = 8883
rootCAPath = "root-CA.crt"
privateKeyPath = "mbus-collector.private.key"
certificatePath = "mbus-collector.cert.pem"

# Keep dictionary of sensor specific variables
# Prefixes are determined by VIF in M-Bus package
# [location, last_min_pressure_VIF, last_max_pressure_VIF, last_inst_pressure_VIF
# [location, last_flow1_VIF, last_flow2_VIF, last_temp_VIF, last_flow1_calc, last_flow2_calc]
sensor_info_dict = {
    "770004242c2d": ["loc-1", "69", "69", "69"], # PressureSensor
    "688268302c2d": ["loc-1", "13", "13", "67", -1, -1], # flowIQ
    "50902542ce9a": ["loc-2", "69", "69", "69"], # Simulated PressureSensor
    "51705369ce9a": ["loc-2", "13", "13", "67", -1, -1], # Simulated flowIQ
    "51705518ce9a": ["loc-3", "69", "69", "69"], # Simulated PressureSensor
    "51705538ce9a": ["loc-3", "13", "13", "67", -1, -1], # Simulated flowIQ
    "50902294ce9a": ["loc-4", "69", "69", "69"], # Simulated PressureSensor
    "51705516ce9a": ["loc-4", "13", "13", "67", -1, -1], # Simulated flowIQ
}

```

```
44  ##############################################################################
45  # Main function
46  ##############################################################################
47  def main():
48      # print_ports()
49      log_port("COM4")
50
51
52  ##############################################################################
53  # Functions
54  ##############################################################################
55
56
57  def print_ports():
58      """
59      Print available data ports
60      """
61      ports = list(list_ports.comports())
62      for p in ports:
63          print(p)
64
65
66  def calculate_pressure(VIF, D1, D2):
67      """
68      Calculate pressure on M-bus format
69      """
70      # Extract bit 1 and 2 from hex number located at temp_pac[22]
71      prefix_bin = bin(int(VIF, 16))[2:].zfill(8)[6:9]
72      prefix = 10 ** (int(prefix_bin, 2) - 3)
73      hex_value = D1 + D2
74      dec_value = int(hex_value, 16) * prefix
75      return round(dec_value, 2)
76
77
78  def calculate_volume(VIF, D1, D2, D3, D4):
79      """
80      Calculate volume on M-bus format
81      """
82      # Extract bit 1 and 2 from hex number located at temp_pac[23]
83      prefix_bin = bin(int(VIF, 16))[2:].zfill(8)[5:9]
84      prefix = 10 ** (int(prefix_bin, 2) - 6)
85      hex_value = D1 + D2 + D3 + D4
86      int_value = int(hex_value, 16) * prefix
87      return round(int_value, 3)
88
89
90  def calculate_temperature(VIF, D1):
91      """
92      Calculate temperature on M-bus format
93      """
94      # Extract bit 1 and 2 from hex number located at temp_pac[22]
95      prefix_bin = bin(int(VIF, 16))[2:].zfill(8)[6:9]
96      prefix = 10 ** (int(prefix_bin, 2) - 3)
97      hex_value = D1
98      #print("variables are ", str(hex_value), " and ", str(prefix))
99      int_value = int(hex_value, 16) * prefix
100     #print("returning ", str(int_value))
101     return int(int_value)
102
103 def calculate_pressure_packet(pac_list, i1, i2, i3):
104     device_name = pac_list[8] + pac_list[7] + pac_list[6] + pac_list[5] + pac_list[4] +
          pac_list[3]
105
106     # [location, last_min_pressure_VIF, last_max_pressure_VIF, last_inst_pressure_VIF
107     global sensor_info_dict
```

```
    last_min_pressure_VIF = sensor_info_dict[device_name][1]
    last_max_pressure_VIF = sensor_info_dict[device_name][2]
    last_inst_pressure_VIF = sensor_info_dict[device_name][3]

    # Min pressure
    press_min_calc = calculate_pressure(
        last_min_pressure_VIF, pac_list[i1], pac_list[i1 - 1]
    )
    pressure_pac = ";;;;;;" + str(press_min_calc)

    # Max pressure
    press_max_calc = calculate_pressure(
        last_max_pressure_VIF, pac_list[i2], pac_list[i2 - 1]
    )
    pressure_pac += ";" + str(press_max_calc)

    # Instant pressure
    press_inst_calc = calculate_pressure(
        last_inst_pressure_VIF, pac_list[i3], pac_list[i3 - 1]
    )
    pressure_pac += ";" + str(press_inst_calc)
    return pressure_pac

def calculate_flow_packet(pac_list, i1, i2, i3):
    device_name = pac_list[8] + pac_list[7] + pac_list[6] + pac_list[5] + pac_list[4] +
        pac_list[3]
    # [location, last_flow1_VIF, last_flow2_VIF, last_temp_VIF, last_flow1_calc,
        last_flow2_calc]
    global sensor_info_dict
    last_flow1_VIF = sensor_info_dict[device_name][1]
    last_flow2_VIF = sensor_info_dict[device_name][2]
    last_temp_VIF = sensor_info_dict[device_name][3]
    last_flow1_calc = sensor_info_dict[device_name][4]
    last_flow2_calc = sensor_info_dict[device_name][5]
    # flow 1
    volume1_calc = calculate_volume(
        last_flow1_VIF,
        pac_list[i1],
        pac_list[i1 - 1],
        pac_list[i1 - 2],
        pac_list[i1 - 3],
    )
    flow_pac = ";" + str(volume1_calc)

    # flow 2
    volume2_calc = calculate_volume(
        last_flow2_VIF,
        pac_list[i2],
        pac_list[i2 - 1],
        pac_list[i2 - 2],
        pac_list[i2 - 3],
    )
    flow_pac += ";" + str(volume2_calc)

    # temperature
    #print("lets calculate som temperature with ", str(i3))
    temp_calc = calculate_temperature(last_temp_VIF, pac_list[i3])
    flow_pac += ";" + str(temp_calc)

    # diff 1 & 2
    if last_flow1_calc == -1:
        flow_pac += ";" + "0"
        flow_pac += ";" + "0"
        sensor_info_dict[device_name][4] = volume1_calc
        sensor_info_dict[device_name][5] = volume2_calc
```

```
171     else:
172         flow1_diff = int(1000 * volume1_calc) - int(1000 * last_flow1_calc)
173         flow2_diff = int(1000 * volume2_calc) - int(1000 * last_flow2_calc)
174         flow_pac += ";" + str(flow1_diff)
175         flow_pac += ";" + str(flow2_diff)
176         sensor_info_dict[device_name][4] = volume1_calc
177         sensor_info_dict[device_name][5] = volume2_calc
178     flow_pac += ";;;"
179     return flow_pac
180 
181 def format_packet(pac):
182     """
183     Format packets received on M-bus format into data that is readable
184     """
185     global sensor_info_dict
186 
187     temp_pac = pac.split(";")
188     device_name = temp_pac[8] + temp_pac[7] + temp_pac[6] + temp_pac[5] + temp_pac[4] +
          temp_pac[3]
189 
190     new_pac = ""
191 
192     # Time package was received
193     new_pac += time.strftime("%H:%M:%S", time.gmtime(int(temp_pac[0])))
194 
195     packet_type = temp_pac[10]
196     man_id_1 = temp_pac[3]
197     man_id_2 = temp_pac[4]
198 
199     if man_id_1 == "2d" and man_id_2 == "2c" and packet_type == "16": # Kamstrup flowIQ
200         if temp_pac[20] == "78": # VIF is transmitted
201             # last_flow1_VIF
202             sensor_info_dict[device_name][1] = temp_pac[27]
203             # last_flow2_VIF
204             sensor_info_dict[device_name][2] = temp_pac[33]
205             # last_temp_VIF
206             sensor_info_dict[device_name][3] = temp_pac[39]
207             i1 = 31
208             i2 = 37
209             i3 = 40
210         elif temp_pac[20] == "79": # VIF is not transmitted
211             i1 = 30
212             i2 = 34
213             i3 = 35
214         new_pac += calculate_flow_packet(temp_pac, i1, i2, i3)
215 
216     elif man_id_1 == "2d" and man_id_2 == "2c" and packet_type == "18": # Kamstrup
          PressureSensor
217         if temp_pac[20] == "78": # VIF is transmitted
218             # last_min_pressure_VIF
219             sensor_info_dict[device_name][1] = temp_pac[22]
220             # last_max_pressure_VIF
221             sensor_info_dict[device_name][2] = temp_pac[26]
222             # last_inst_pressure_VIF
223             sensor_info_dict[device_name][3] = temp_pac[30]
224             i1 = 24
225             i2 = 28
226             i3 = 32
227         elif temp_pac[20] == "79": # VIF is not transmitted
228             i1 = 26
229             i2 = 28
230             i3 = 30
231         new_pac += calculate_pressure_packet(temp_pac, i1, i2, i3)
232 
233     elif man_id_1 == "9a" and man_id_2 == "ce" and packet_type == "16": # Simulated flowIQ
```

```
234         i1 = 31
235         i2 = 35
236         i3 = 36
237         new_pac += calculate_flow_packet(temp_pac, i1, i2, i3)
238 
239     elif man_id_1 == "9a" and man_id_2 == "ce" and packet_type == "18": # Simulated
        PressureSensor
240         i1 = 27
241         i2 = 29
242         i3 = 31
243         new_pac += calculate_pressure_packet(temp_pac, i1, i2, i3)
244 
245     else:
246         print("unknown␣packet")
247 
248     rssi_hex = str(pac[-2]) + str(pac[-1])
249     rssi_int = int(rssi_hex, 16)
250     new_pac += ";" + str(rssi_int)
251 
252     return new_pac
253 
254 
255 def init_aws_upload(myAWSIoTMQTTClient):
256     """
257     Initialize AWS uploading
258     """
259     myAWSIoTMQTTClient.configureEndpoint(host, cloud_port)
260     myAWSIoTMQTTClient.configureCredentials(
261         rootCAPath, privateKeyPath, certificatePath
262     )
263 
264     # AWSIoTMQTTClient connection configuration
265     myAWSIoTMQTTClient.configureAutoReconnectBackoffTime(1, 32, 20)
266     myAWSIoTMQTTClient.configureOfflinePublishQueueing(-1)  # Set as infinite
267     myAWSIoTMQTTClient.configureDrainingFrequency(2)  # Draining: 2 Hz
268     myAWSIoTMQTTClient.configureConnectDisconnectTimeout(10)  # 10 sec
269     myAWSIoTMQTTClient.configureMQTTOperationTimeout(5)  # 5 sec
270 
271     # Connect to AWS IoT
272     myAWSIoTMQTTClient.connect()
273 
274 
275 def print_packet(packet):
276     """
277     Print a data packet
278     """
279     packet_list = packet.split(";")
280     for i in packet_list:
281         print(i, end="\t")
282     print("␣")
283 
284 
285 def save_packet(save_loc, packet):
286     """
287     Save a data packet in a csv format
288     """
289     with open(save_loc, "a", newline="") as f:
290         writer = csv.writer(f, delimiter=",")
291         writer.writerow([packet])
292 
293 
294 def log_port(port):
295     """
296     Read serial port and optionally save the data to file and cloud
297     """
```

```
298     myAWSIoTMQTTClient = None
299     myAWSIoTMQTTClient = AWSIoTMQTTClient(clientId)
300     init_aws_upload(myAWSIoTMQTTClient)
301
302     ser = serial.Serial(port, 19200)  # open serial port.
303     print(ser.name)
304     ser.reset_input_buffer()  # Discard all content of input buffer
305
306     while True:
307         try:
308             ######################################################################
309             # Read all packets
310             ######################################################################
311             device_name = ""
312             ser_byte = ser.read()  # Read first byte to determine length
313             in_hex = ser_byte.hex()  # Convert to hex
314             packet = in_hex
315
316             # Read the rest of the bytes
317             for i in range(int(in_hex, 16)):
318                 ser_byte = ser.read()
319                 in_hex = ser_byte.hex()
320                 packet += ";" + in_hex
321
322                 if i <= 6 and i > 0:  # Store all bytes for device name
323                     device_name = in_hex + device_name
324
325             # Time and date calculation
326             date_today = datetime.today().strftime("%Y-%m-%d")
327             now = datetime.now()
328             seconds_since_midnight = int(
329                 (
330                     now - now.replace(hour=0, minute=0, second=0, microsecond=0)
331                 ).total_seconds()
332             )
333
334             timed_packet = str(seconds_since_midnight) + ";" + packet
335
336             #if timed_packet.split(";")[3] != "2d":
337             #    continue
338
339             ######################################################################
340             # Format data
341             ######################################################################
342             formatted_packet = format_packet(timed_packet)
343             print_packet(device_name + ";" + formatted_packet)
344
345             ######################################################################
346             # Save raw data to file
347             ######################################################################
348             raw_save_loc = device_name + "-" + date_today + ".csv"
349             save_packet(raw_save_loc, timed_packet)
350
351             ######################################################################
352             # Save formatted data to file
353             ######################################################################
354             formatted_save_loc = sensor_info_dict[device_name][0] + "_" + date_today +
                    "-formatted.csv"
355             save_packet(formatted_save_loc, formatted_packet)
356
357             ######################################################################
358             # Save formatted data to cloud
359             ######################################################################
360             formatted_packet_list = formatted_packet.split(";")
361             timed_packet_list = timed_packet.split(";")
```

```
            sensor_type = "Unknown"
            data_to_upload = {}

            # Define data to be uploaded
            if timed_packet_list[10] == "16":
                sensor_type = "flow"
                data_to_upload = {
                    #"Date": date_today + " " + formatted_packet_list[0],
                    "SerialNumber": device_name,
                    "CollectorID": clientId,
                    "Location": sensor_info_dict[device_name][0],
                    "flow_inst": float(formatted_packet_list[1]),
                    "flow_max_month": float(formatted_packet_list[2]),
                    "temp_ambient": int(formatted_packet_list[3]),
                    "flow_inst_diff": int(formatted_packet_list[4]),
                    "flow_max_month_diff": int(formatted_packet_list[5]),
                    "RSSI": int(formatted_packet_list[9]),
                }
            elif timed_packet_list[10] == "18":
                sensor_type = "pressure"
                data_to_upload = {
                    #"Date": date_today + " " + formatted_packet_list[0],
                    "SerialNumber": device_name,
                    "CollectorID": clientId,
                    "Location": sensor_info_dict[device_name][0],
                    "min_pressure": float(formatted_packet_list[6]),
                    "max_pressure": float(formatted_packet_list[7]),
                    "inst_pressure": float(formatted_packet_list[8]),
                    "RSSI": int(formatted_packet_list[9]),
                }

            # Define topic name
            topic = "collectors/" + clientId + "/" + sensor_type + "/" + device_name
            messageJson = json.dumps(data_to_upload)
            try:
                myAWSIoTMQTTClient.publish(topic, messageJson, 1)
                # print('Published topic %s: %s\n' % (topic, messageJson))
            except Exception as e:
                print("Error:␣", e)
            ###############################################################

        except Exception as e:
            error_time = datetime.today().strftime("%Y-%m-%d/%H:%M:%S")
            print("Error␣occured␣at", error_time)
            with open("error_log.csv", "a", newline="") as f:
                writer = csv.writer(f, delimiter=",")
                writer.writerow([error_time])
                writer.writerow([e])
            print("Continuing␣logging...")
            continue


if __name__ == "__main__":
    main()
```

Code listing B.1: Serial port logging code used for several tasks for the gateway. It reads values received on a serial port, it formats the values to a readable format, it stores the data locally and it uploads the data to IoT core.

Appendix C

CSV to IBM hex format converter

```
"""
================================================================================
Program to convert csv data to intel hex file format
================================================================================
Hex File Format:

:llaaaatt[dd...]cc

   :    is the colon that starts every Intel HEX record.
   ll   is the record-length field that represents the number of data bytes (dd)
        in the record.
   aaaa is the address field that represents the starting address for subsequent
        data in the record.
   tt   is the field that represents the HEX record type, which may be one of
        the following:
   00 - data record
   01 - end-of-file record
   02 - extended segment address record
   04 - extended linear address record
   05 - start linear address record (MDK-ARM only)
   dd   is a data field that represents one byte of data. A record may have
        multiple data bytes. The number of data bytes in the record must match
        the number specified by the ll field.
   cc   is the checksum field that represents the checksum of the record. The
        checksum is calculated by summing the values of all hexadecimal digit
        pairs in the record modulo 256 and taking the two's complement.
"""

from intelhex import IntelHex
import csv

hex_file = "test_data_hex.hex"
csv_file = "770004242c2d-2021-03-08_14.csv"

ih = IntelHex()

base_address = "0x8000"
base_address_int = int(base_address, 16)

with open(csv_file) as f:
   csv_reader = csv.reader(f, delimiter=';')
   line_count = 0
   el_count = 0
   for row in csv_reader:
      for el in row:
```

```
46          el = int(el, 16)
47          #print(el)
48          ih[base_address_int + el_count] = el
49          el_count += 1
50      line_count += 1
51
52  print("Elements:", el_count)
53  print("lines:", line_count)
54  print("Base␣Address:", base_address)
55
56  ih.write_hex_file(hex_file)
```

Code listing C.1: Script used to format CSV data to a HEX file to be used by Silicon Labs Flash Programmer.

Appendix D

Data Analysis

```
"""
This script has two functions:
- Generate a water demand pattern based on real volume measurements
- Resample pressure measurements to a desired interval, to be used for generating pressure residuals
"""
import matplotlib.pyplot as plt
import pandas as pd
import matplotlib.dates as mdates


def head2bar(head):
    return head * 0.09804

def bar2head(bar):
    return bar * 10.1974

def get_interpolated_pressure(source):
    """
    Get resampled values of pressure
    """
    df = pd.read_csv(source, delimiter=';', parse_dates=['Time'], index_col='Time')
    pressure_upsample = df['Press␣Inst␣[bar]'].resample('1T').mean()
    pressure_interpolate = pressure_upsample.interpolate()
    pressure_downsample = pressure_interpolate.resample('10T').mean()
    return pressure_downsample

def get_interpolated_flow(source):
    """
    Get resampled values of flow
    """
    df = pd.read_csv(source, delimiter=';', parse_dates=['Time'], index_col='Time')
    flow_upsample = df['Flow␣1␣[m^3]'].resample('1T').mean()
    flow_interpolate = flow_upsample.interpolate()
    flow_downsample = flow_interpolate.resample('10T').mean()
    return flow_downsample


def get_flow_rate(df):
    """
    Get the average flow for a time range, based on measured volume.
    """
    flow_rate_frame = pd.DataFrame.copy(df)
    flow_rate_frame.iloc[0] = 0
    for i in range(1, len(df)):
```

```
45          # Convert from m^3 to L with 1000. Should be divided by 600 for 10 minutes
46          flow_rate_frame.iloc[i] = 120*(1000/600)*df.iloc[i] - 120*(1000/600)*df.iloc[i-1]
47      return flow_rate_frame
48
49  def graph_dataframes(dfs):
50      """
51      Graph all dataframes (dfs) that is set as input
52      """
53      first_frame = dfs[0]
54      ax = first_frame.plot(kind = 'line',
55                    title='Water Demand Pattern for 2021-03-06',
56                    figsize=(16,5))
57
58      for i in range(1, len(dfs)):
59          dfs[i].plot(ax=ax,
60                  color='r',
61                  kind = 'line',
62                  grid=True,
63                  label='Estimated Flow Rate [LPS]')
64      ax.legend()
65      ax.xaxis.set_major_locator(mdates.MinuteLocator(interval=60))
66      ax.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M:%S"))
67      plt.show()
68
69  def print_frame(df):
70      """
71      Set options for the printing of dataframes
72      """
73      with pd.option_context('display.max_rows', None, 'display.max_columns', None):  # more
            options can be specified also
74          print(df)
75
76  def main():
77      """
78      - Save interpolated pressure for pressure residual generation and flow rate for EPANET model
79      - Graph flow rate and volume measurements to see difference
80      """
81      source_location = "../../data/2021/03-mar/"
82      save_location = "../../data/pressure_residual_calculation/"
83      data_file = source_location + '2021-03-02-formatted.csv'
84
85      # Load dataframes
86      df = pd.read_csv(data_file, delimiter=';', parse_dates=['Time'], index_col='Time')
87      df_press = pd.DataFrame(df, columns=['Press Inst [bar]'])
88      df_flow = pd.DataFrame(df, columns=['Flow 1 [m^3]'])
89      df_diff = pd.DataFrame(df, columns=['Diff 1 [L]'])
90
91      # Data Preprocessing
92      df_press = df_press.dropna()
93      df_flow = df_flow.dropna()
94      df_diff = df_diff.rename(columns={'Diff 1 [L]': 'Measured Volume [L]'}).dropna()
95
96      # Process dataframes
97      df_press_i = get_interpolated_pressure(data_file)
98      df_flow_i = get_interpolated_flow(data_file)
99      df_flow_rate = get_flow_rate(df_flow_i)
100
101     # Save pressure and flow rate
102     df_merge = pd.merge(df_press_i, df_flow_rate, on='Time')
103     df_merge.to_excel(save_location + 'out.xlsx')
104
105     # Graph dataframes in dfs
106     dfs = [df_diff, df_flow_rate]
107     graph_dataframes(dfs)
108
```

```
109 if __name__ == "__main__":
110     main()
```

Code listing D.1: Script used to resample flow and pressure data, as well as estimate flow from volume measurements.

Appendix E

Simulated Meter Code

```
 /**
  @file       main_meter.c
  @copyright  STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
  @author     STACKFORCE
  @brief      Demo application of a meter device using the application layer.

              Demo of point to point communication.
*/

/*============================================================================*/

/*============================================================================
                            INCLUDE FILES
============================================================================*/
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\utils\wmbus_clock_api.h"
#include "inc\pub\utils\wmbus_api.h"
#include "inc\prv\cfg\wmbus_config.h"
#include "inc\pub\dll\wmbus_dll_defines.h"
#include "inc\pub\tpl\wmbus_tpl_api.h"
#include "inc\pub\utils\wmbus_timer_api.h"
#include "inc\pub\hal\wmbus_hal.h"

#include "em_gpio.h"
#include "em_cmu.h"

/*============================================================================
                            DEFINES
============================================================================*/
#ifndef WMBUS_CFG_DEVICE
#error Please define the device configuration to a METER device!
#elif WMBUS_CFG_DEVICE != WMBUS_CFG_DEVICE_METER
#error Please define the device configuration to a METER device!
#endif /* WMBUS_CFG_DEVICE */

/*======================== CLOCK ============================================*/
/* 7 Extracts the fields of the date format I. */
/*! Bit 1..6 */
#define CLOCK_SECOND_GET(x)       ((x)[5U] & 0x3FU)
#define CLOCK_SECOND_SET(x,y)     (x)[5U] &= ~(0x3FU); (x)[5U] += (y)

/*! Bit 9..14 */
#define CLOCK_MINUTE_GET(x)       ((x)[4U] & 0x3FU)
#define CLOCK_MINUTE_SET(x,y)     (x)[4U] &= ~(0x3FU); (x)[4U] += (y)

```

```
/*! Bit 17..21 */
#define CLOCK_HOUR_GET(x)         ((x)[3U] & 0x1FU)
#define CLOCK_HOUR_SET(x,y)       (x)[3U] &= ~(0x1FU); (x)[3U] += (y)

/*! CLOCK_DAY_MAX is defined by CLOCK_GET_DAYS_OF_MONTH */
#define CLOCK_DAY_GET(x)          ((x)[2U] & 0x1FU)
#define CLOCK_DAY_SET(x,y)        (x)[2U] &= ~(0x1FU); (x)[2U] += (y)

/*! Bit 33..36, 0 = not specified */
#define CLOCK_MONTH_GET(x)        ((x)[1U] & 0x0FU)
#define CLOCK_MONTH_SET(x,y)      (x)[1U] &= ~(0x0FU); (x)[1U] += (y)

/*! Bit 30..32+37..40, 0x7F = not specified */
#define APL_CLOCK_YEAR_GET(x)         ((((x)[2U] & 0xE0U) >> 5U) + \
                                        (((x)[1U] & 0xF0U) >> 1U))


#define APL_CLOCK_YEAR_SET(x,y)    (x)[1U] &= (0xFU); \
                                        (x)[1U] += ((((y) % 100U) & 0x78U) << 1U); \
                                          (x)[2U] &= (0x1FU); \
                                            (x)[2U] += ((((y) % 100U) & 0x7U) << 5U); \

/*! Bit 22..24, 0 = not specified, 1 = Monday, 7 = Sunday. */
#define CLOCK_DAY_OF_WEEK_NULL    0U

/*! Bit 41..46, 0 = not specified */
#define CLOCK_WEEK_NULL           0U

/*! Bit 22..24, 0 = not specified, 1 = Monday, 7 = Sunday. */
#define CLOCK_DAY_OF_WEEK_GET(x)  (((x)[3U] & 0xE0U) >> 5U)
#define CLOCK_DAY_OF_WEEK_SET(x,y)  (x)[3U] &= ~(0xE0U); (x)[3U] += ((y) << 5U)

/*! @todo Weeks are not provided yet. */
/*! Bit 41..46, 0 = not specified */
#define CLOCK_WEEK_GET(x)         ((x)[0U] & 0x3FU)
#define CLOCK_WEEK_SET(x,y)       (x)[0U] &= ~(0x3FU); (x)[4U] += (y)

/*======================== DIF and VIF=====================================*/
/*! Instaneous value */
#define DIF_FUNC_INSTANEOUS                0x00U
/*! 48 Bit IntegerBinary */
#define DIF_DATA_FIELD_48_INT              0x06U
/*! E110 1101 -> data field 0100b, type F
    E110 1101 -> data field 0011b, type J
    E110 1101 -> data field 0110b, type I */
#define VIF_DATE_TIME                      0x6DU


/*======================== DIF and VIF=====================================*/

#define BYTE_SIZE 8

/*============================================================================
                            ENUMS
============================================================================*/

/*============================================================================
                            VARIABLES
============================================================================*/
/* Collector and meter address */
/* example meter address */
s_wmbus_addr_t gs_addr = {{0xce,0x9a},        /* Manufacturer (here STZ) */
                    {0x51,0x70,0x55,0x18},    /* ident number            */
                    0x1d,                     /* version                 */
                    WMBUS_DEV_TYPE_PRESSURE};    /* type, here water        */
```

```
/* WMBUS_DEV_TYPE_COLD_WATER 0x16*/
/* WMBUS_DEV_TYPE_PRESSURE 0x18*/

/* example collector address */
s_wmbus_addr_t gs_collector = {{0xce,0x9a},   /* Manufacturer (here STZ) */
                    {0x80,0x00,0x00,0x02},    /* ident number            */
                     0x23,                    /* version                 */
                    WMBUS_DEV_TYPE_OTHER};    /* type, here other        */

uint8_t gpc_key[] = {0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,
                    0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF};

s_tpl_startAttr_t gs_start_attr =
{
  /* Init start strucutre */
  /* Frequency offset for the carrier. */
  0,
  /* Device address. */
  &gs_addr,
  /* Collector address. */
  &gs_collector,
  /* Set the device to connected. Only if the device is a meter device.
      Otherwise this field is ignored. */
  TRUE,
  /* Periodical interval for sending data. Only if the device is a meter
      device. [ms] */
  3000U,
  /*! Encryption key */
  gpc_key,
};

/* Global acc-number */
uint8_t gc_acc;

/* Global day prefix. 0-6 for mon to sun */
#define DAY_PREFIX      1
/* Global hour prefix. (0-23) */
#define HOUR_PREFIX     12
/* Global packet-number */
uint32_t curr_packet = (DAY_PREFIX*24*60*60+HOUR_PREFIX*60*60)/95;

/*! Clock. */
s_clock_t gs_clock;
/*============================================================================
                            FUNCTION PROTOTYPES
============================================================================*/
void loc_clockCreateData(uint8_t* pc_dst, s_clock_t* ps_src);
/*============================================================================
                            FUNCTIONS
============================================================================*/

/*============================================================================*/
/*!
 * @brief  Converts the time of ps_src to telegram format.
 * @param pc_dst  Destination memory.
 * @param ps_src  Clock source.
 */
/*============================================================================*/
void loc_clockCreateData(uint8_t* pc_dst, s_clock_t* ps_src)
{
  if((pc_dst != NULL) && (ps_src != NULL))
  {
    CLOCK_SECOND_SET(pc_dst,      ps_src->c_seconds);
    CLOCK_MINUTE_SET(pc_dst,      ps_src->c_minutes);
    CLOCK_HOUR_SET(pc_dst,        ps_src->c_hours);
```

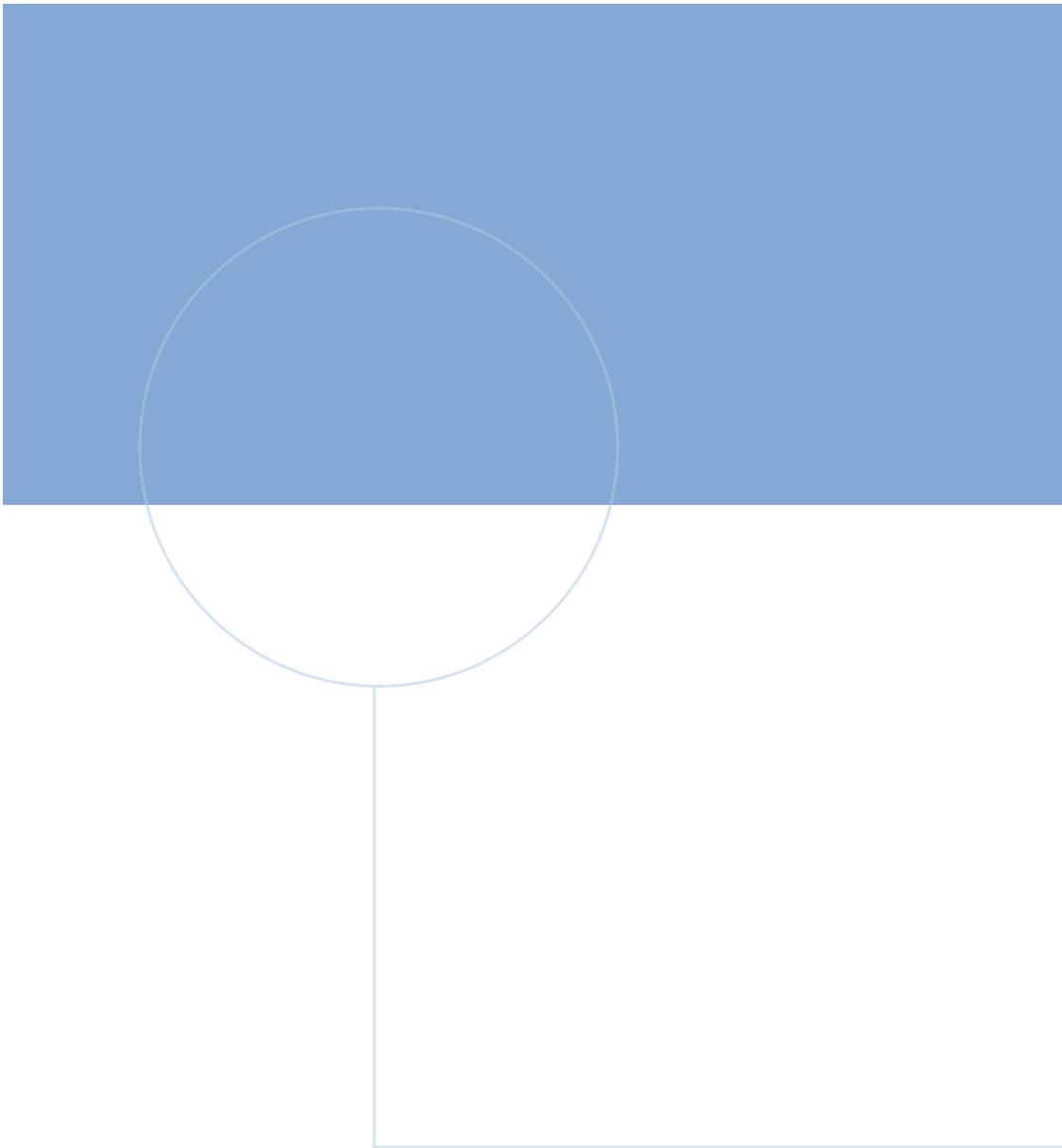
```
176     CLOCK_DAY_SET(pc_dst,         ps_src->c_days);
177     CLOCK_MONTH_SET(pc_dst,       ps_src->c_months);
178     APL_CLOCK_YEAR_SET(pc_dst,        ps_src->i_years);
179     CLOCK_DAY_OF_WEEK_SET(pc_dst, CLOCK_DAY_OF_WEEK_NULL);
180     CLOCK_WEEK_SET(pc_dst,        CLOCK_WEEK_NULL);
181   } /* if */
182 } /* loc_clockCreateData() */
183 /*============================================================================*/
184 /*!
185  * @brief Main program.
186  */
187 /*============================================================================*/
188 void main(void)
189 {
190   /* Initialises the hal. */
191   if(wmbus_hal_init() == E_HAL_STATUS_SUCCESS)
192   {
193     /* Initialize LED pins */
194     CMU_ClockEnable(cmuClock_HFPER, true);
195     CMU_ClockEnable(cmuClock_GPIO, true);
196     GPIO_PinModeSet(gpioPortF, 6, gpioModePushPull, 0);
197     GPIO_PinModeSet(gpioPortF, 7, gpioModePushPull, 0);
198     GPIO_PinOutSet(gpioPortF, 6);
199     /* initialize tpl */
200     wmbus_tpl_init();
201     /* Initialisation of the clock */
202     wmbus_clock_init(&gs_clock);
203 
204     /* wm-bus settings */
205     wmbus_tpl_setInterval(95000U);
206     wmbus_tpl_setAddrOwn(&gs_addr);
207 
208     /* start the tpl */
209     wmbus_tpl_start(&gs_start_attr);
210     /* start the clock */
211     wmbus_clock_start(&gs_clock);
212 
213     while(TRUE)
214     {
215       /* run the tpl layer */
216       wmbus_tpl_run();
217 
218       /* runs the clobal clock */
219       wmbus_clock_run(&gs_clock);
220 
221     } /* while */
222   }
223 } /* main() */
224 
225 /*============================================================================*/
226 /*! wmbus_tpl_evt_sendUserData() */
227 /*============================================================================*/
228 
229 
230 
231 /*============================================================================*/
232 /*! wmbus_tpl_evt_sendUserData() */
233 /*============================================================================*/
234 void wmbus_tpl_evt_mtr_sendUserData(void)
235 {
236   uint32_t packets = 6282;
237   uint32_t* start_addr = (uint32_t*)(FLASH_BASE + 2048*16); //location 0x8000
238   uint32_t* curr_addr = (uint32_t*)(start_addr + curr_packet * 4);
239 
240   if(curr_packet != packets-1)
```

```
241     curr_packet++;
242   else
243     curr_packet = 0;
244 
245   uint32_t sensor_data = *curr_addr;
246   uint8_t word_part0 = sensor_data >> (0 * BYTE_SIZE);
247   uint8_t word_part1 = sensor_data >> (1 * BYTE_SIZE);
248   uint8_t word_part2 = sensor_data >> (2 * BYTE_SIZE);
249   uint8_t word_part3 = sensor_data >> (3 * BYTE_SIZE);
250 
251   sensor_data = *(curr_addr + 1);
252   uint8_t word_part4 = sensor_data >> (0 * BYTE_SIZE);
253   uint8_t word_part5 = sensor_data >> (1 * BYTE_SIZE);
254   uint8_t word_part6 = sensor_data >> (2 * BYTE_SIZE);
255   uint8_t word_part7 = sensor_data >> (3 * BYTE_SIZE);
256 
257   sensor_data = *(curr_addr + 2);
258   uint8_t word_part8 = sensor_data >> (0 * BYTE_SIZE);
259   uint8_t word_part9 = sensor_data >> (1 * BYTE_SIZE);
260   uint8_t word_part10 = sensor_data >> (2 * BYTE_SIZE);
261   uint8_t word_part11 = sensor_data >> (3 * BYTE_SIZE);
262 
263   sensor_data = *(curr_addr + 3);
264   uint8_t word_part12 = sensor_data >> (0 * BYTE_SIZE);
265   uint8_t word_part13 = sensor_data >> (1 * BYTE_SIZE);
266   uint8_t word_part14 = sensor_data >> (2 * BYTE_SIZE);
267   uint8_t word_part15 = sensor_data >> (3 * BYTE_SIZE);
268 
269   /* The periodical data includes the time. Further data can be added here
270    * to the telegram:
271    * Record 0: time information (already set automatically!)
272    * Record 1: our example data (pc_staticData[])
273    */
274   uint8_t pc_staticData[]={ word_part0,
275                             word_part1,
276                             word_part2,
277                             word_part3,
278                             word_part4,
279                             word_part5,
280                             word_part6,
281                             word_part7,
282                             word_part8,
283                             word_part9,
284                             word_part10,
285                             word_part11,
286                             word_part12,
287                             word_part13,
288                             word_part14,
289                             word_part15};
290   /* Header of the meter telegram */
291   s_tpl_headerShort_t  s_headerShort;
292   /* Timestamp data to append. */
293   uint8_t pc_data[6U];
294   /* ID of the created telegram */
295   uint8_t c_tlgId;
296   /* boolean which cheks if the telegram must be deleted */
297   bool_t b_deleteTelegram = TRUE;
298 
299   /* Creates the telegram. */
300   c_tlgId = wmbus_tpl_createTlg(DLL_FIELD_C_PRM_UD_NOREPL,  /* User data / no replay */
301                                 NULL,                       /* Use default meter address */
302                                 TPL_FIELD_CI_HEADER_SHORT);
303   if(c_tlgId != DLL_ERR_TLG_NOT_AVAILABLE)
304   {
305     s_headerShort.e_type = E_TPL_HEADER_TYPE_SHORT;
```

```
306     s_headerShort.c_accNo = gc_acc;
307     s_headerShort.c_status = 0x00;
308     s_headerShort.i_signature = 0x8500;
309
310     wmbus_tpl_setHeader(c_tlgId, (s_tpl_header_t*)&s_headerShort);
311
312     wmbus_tpl_encryptPrepare(c_tlgId);
313
314     /* Add the date to the telegram */
315     *pc_data = DIF_FUNC_INSTANEOUS + DIF_DATA_FIELD_48_INT;
316     if(wmbus_tpl_writeTlg(c_tlgId, pc_data, 1U, DLL_TLG_WRITE_APPEND, TRUE) == TRUE)
317     {
318       /* Value information field (VIF) */
319       *pc_data = VIF_DATE_TIME;
320
321       if(wmbus_tpl_writeTlg(c_tlgId, pc_data, 1U, DLL_TLG_WRITE_APPEND, TRUE) == TRUE)
322       {
323         /* Data */
324         loc_clockCreateData(pc_data, &gs_clock);
325
326         wmbus_tpl_writeTlg(c_tlgId, pc_data, 6U, DLL_TLG_WRITE_APPEND, TRUE);
327
328         wmbus_tpl_writeTlg(c_tlgId, pc_staticData, sizeof(pc_staticData), DLL_TLG_WRITE_APPEND
                ,FALSE);
329
330         if(wmbus_tpl_encrypt(c_tlgId) == E_TPL_CRYPT_RET_OK)
331         {
332           /* Sends the telegram. */
333           if(wmbus_tpl_sendTlg(c_tlgId))
334           {
335             /* The telelgramm was send successfully */
336             b_deleteTelegram = FALSE;
337             if(gc_acc != 0xFF)
338               gc_acc++;
339             else
340               gc_acc = 0;
341           }/* if */
342         }/* if */
343       }/* if */
344     }/* if */
345     if(b_deleteTelegram == TRUE)
346     {
347       wmbus_tpl_destroyTlg(c_tlgId);
348     }/* if */
349   }/* if */
350 } /* wmbus_tpl_evt_sendUserData() */
351
352 /*============================================================================*/
353 /*! wmbus_tpl_evt_tx() */
354 /*============================================================================*/
355 void wmbus_tpl_evt_mtr_tx(uint8_t c_tlgId)
356 {
357   /*
358    * Whenever a radio telegram is sent this event is called. You can use it
359    * to toggle LEDs or count transmitted telegrams.
360    * Function can be disabled using APL_EVT_TX_ENABLED in "wmbus_global.h".
361    */
362   GPIO_PinOutToggle(gpioPortF, 6);
363   GPIO_PinOutToggle(gpioPortF, 7);
364
365 }/* wmbus_tpl_evt_tx() */
366
367 /*============================================================================*/
368 /*! wmbus_tpl_evt_getCiHeader() */
369 /*============================================================================*/
```

```
370 E_TPL_HEADER_TYPE_t wmbus_tpl_evt_getCiHeader(uint8_t c_ci)
371 {
372   E_TPL_HEADER_TYPE_t e_return = E_TPL_HEADER_TYPE_INVALID;
373 
374   switch(c_ci)
375   {
376     case 0xA1U:   /* STACKFORCE specific: Transmit string with no header. */
377       e_return = E_TPL_HEADER_TYPE_NO;
378       break;
379     case 0xA2U:   /* STACKFORCE specific: Transmit string with short header. */
380       e_return = E_TPL_HEADER_TYPE_SHORT;
381       break;
382     case 0xA3U:   /* STACKFORCE specific: Transmit string with long header. */
383       e_return = E_TPL_HEADER_TYPE_LONG;
384       break;
385     case 0xA4U:   /* STACKFORCE specific: Transmit string with short header. */
386       e_return = E_TPL_HEADER_TYPE_SHORT;
387       break;
388     default:      /* The application does not know the CI field. */
389       e_return = E_TPL_HEADER_TYPE_INVALID;
390       break;
391 
392       /* please insert specific CI fields here */
393 
394   } /* switch(c_ci) */
395 
396   return e_return;
397 } /* wmbus_tpl_evt_getCiHeader() */
398 
399 /*============================================================================*/
400 /*! wmbus_tpl_evt_tlgAvailable() */
401 /*============================================================================*/
402 void wmbus_tpl_evt_mtr_tlgAvailable(E_WMBUS_RX_t e_status, uint8_t c_tlgReqId,
403                           uint8_t c_tlgId)
404 {
405   /* delete telegram after handling it */
406   wmbus_tpl_destroyTlg(c_tlgId);
407 } /* wmbus_tpl_evt_tlgAvailable() */
```

Code listing E.1: Main code file used for programming simulated meters. This one is for simulated pressure meters. A slight modification is done for simulateed flow meters, as shown in code listing 5.1.



NTNU

Norwegian University of
Science and Technology