University of Zurich[UZH]

ASSIGNMENT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Database Solution for Offline Graphical Visualization of Sensor Data

Christian Ott
Zürich, Switzerland
Student ID: 12-723-896

Supervisor: Dr. Corinna Schmitt
Date of Submission: January 10, 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Assignment
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

This report documents the development process of a database solution for offline graphical visualization of sensor data. The database solution should be integrated into CoMaDa. CoMaDa is a Java framework that offers a graphical user interface for configuration, network management and data handling of wireless sensor networks. The new developed database solution should be integrated in the visualization component of CoMaDa to extend the current functionality. With the database connected to CoMaDa, the visualization component should be able to visualize historical sensor data, contrary to the current state, where only real-time sensor data could be visualized. It was decided to implement a database inside the CoMaDa environment, using PostgreSQL as database technology. The database schema was reused from an already existing database in WebMaDa, an extension to CoMaDa offering mobile access to the sensor networks. The integration of the database into CoMaDa framework was done in a flexible way using a database abstraction layer. The abstraction layer decouples CoMaDa from the used database technology. The existing graphical visualization solution was optimized and adapted to use the database as data source. The results of the implementation were then evaluated regarding performance and usability.

Zusammenfassung

Diese Arbeit präsentiert den Entwicklungsprozess einer Datenbanklösung für eine offline Methode zur graphischen Visualisierung von Sensordaten. Das Ziel war die Integration einer Datenbanklösung in CoMaDa. CoMaDa ist ein Java Framework welches eine benutzerfreundliche graphische Schnittstelle zur Konfiguration, Netzwerkmanagement und Datenhandhabung von drahtlosen Sensornetzwerken anbietet. Die neu entwickelte Datenbanklösung sollte in die graphische Visualisierungskomponente integriert werden, um deren aktuellen Funktionsumfang zu erweitern. Durch die mit CoMaDa verbundene Datenbank soll die Visualisierungskomponente, im Gegensatz zur aktuellen Situation, die Möglichkeit erhalten, historische Messdaten anzuzeigen. Es wurde entschieden eine Datenbank in der CoMaDa Umgebung mit PostgreSQL als Datenbanktechnologie zu implementieren. Das Datenbankschema konnte aus einer, schon existierenden Datenbank in WebMaDa, wiederverwendet werden. WebMaDa ist eine Erweiterung zu CoMaDa und bietet Fernzugriff auf verbundene Sensor Netzwerke. Die Integration der Datenbank in das CoMaDa Framework wurde auf flexible Weise über eine Datenbankabstraktionsschicht umgesetzt. Die Abstraktionsschicht entkoppelt CoMaDa von der genutzten Datenbanktechnologie. Die existierende graphische Visualisierungslösung wurde verbessert und für die Verwendung der Datenbank als Datenquelle angepasst. Die Resultate der Implementation wurden schlussendlich mit Bezug zu Performanz und Gebrauchstauglichkeit ausgewertet.

Acknowledgments

First, I would like to sincerely thank my supervisor Dr. Corinna Schmitt for her continuous support, guidance, patience and her valuable inputs and comments during the last three months.
I would also like to thank Prof. Dr. Burkhard Stiller for the possibility to complete this assignment at the Communications Systems Group at the Department of Informatics of the University of Zurich.

Contents

Abstract **i**

Zusammenfassung **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 2

1.3 Thesis Outline 2

2 Related Work **3**

2.1 SecureWSN 3

2.1.1 CoMaDa 5

2.1.2 WebMaDa 7

2.1.3 Offline Data Visualization for CoMaDa 9

2.2 Database Solutions 10

2.2.1 SQLite 10

2.2.2 MySQL 11

2.2.3 PostgreSQL 11

3 Design Decisions **13**

3.1 Database Decisions 13

3.1.1 External Database 14

3.1.2 Database in WebMaDa 15

3.1.3 Database in CoMaDa 16

3.1.4 Mixed Database in CoMaDa and WebMaDa 17

3.2 Implementation specific Decisions 18

3.2.1 Database Design 18

3.2.2 Database Integration into the Graphical Visualization Component 20

3.3 Summary of Decisions 21

4 Implementation **23**

4.1 Database 23

4.2 CoMaDa Backend Integration 26

4.3 CoMaDa Frontend Integration 32

5 Evaluation **37**

5.1 Improvement possibilities of the SecureWSN Frameworks 37

5.2 Database Solution 39

5.3 Database Integration 40

6 Summary and Conclusions **45**

Bibliography **47**

Abbreviations **51**

List of Figures **51**

List of Tables **53**

List of Listings **55**

A Installation Guidelines **59**

A.1 Database Installation 59

A.2 Configuration Possibilities 60

B Contents of the CD **61**

Chapter 1

Introduction

1.1 Motivation

Due to the growth of the Internet and the device diversity together with their communication capability, the Internet of Things (IoT) determines a highly relevant topic as of today. IoT is not limited to Client-Server (C/S) architectures, Peer-to-Peer (P2P) networks, and well-known devices like server, computer, and routers any more. It especially includes wireless sensor devices connected within a Wireless Sensor Network (WSN) [17].

The application range of those WSNs reaches from intelligent homes via logistics and health care to environmental monitoring. All applications have in common a huge amount of collected sensor data (e.g. temperature, brightness, humidity). In general, this data is stored in a database and accessible over time. One case of analysis is the value development over time. The simplest way to visualize this development is to plot the data in curve diagrams. This can be done using online solutions like Xively [34] or offline solutions like Google Charts [12]. The latter solution is independent of a third party like Xively and offers same functionality except the possibility to access data that was recorded previously, and then stored by Xively. Currently this problem has been partially solved for CoMaDa by writing the data that is used to generate the charts to the session storage [30]. By using session storage, the data is stored until the user closes the tab or window and starts a new session. Although all the recorded sensor data is stored in several text files by CoMaDa [11] on the client machine there exists no solution to store the data client side in a structural or relational manner and thus, there is no possibility to access older data and visualize it.

The main contribution of this thesis to CoMaDa is the implementation of a database solution to improve the shortcomings of the current solution with Google Charts described above. This database integration in conjunction with the already implemented visualization solution finally allows to display historical WSN data.

1.2 Description of Work

The work of this assignment is manifold. In a first step, current existing solutions in CoMaDa 1.1 and WebMaDa 1.1 should be analyzed to gain an understanding of the frameworks. As a result of the first phase, a list of possible improvements should be generated. During the implementation phase a selection of these improvements may be corrected, at least for the adapted parts of the frameworks. Due to the recent additions by various student assignments, functionality was greatly improved, unfortunately this lead to some degree of code fragmentation, and inefficiencies may have been aggregated.
The second phase of this assignment consisted of the design and implementation of a database solution for the sensor data in CoMaDa. The database solution should be integrated into CoMaDa and work as a data source for the offline sensor data visualization method developed by Tim Strasser [30]. The implementation of this database solution will improve the existing CoMaDa environment by adding the possibility to analyze historical senor data.
During the third phase of this assignment, the solution of the second phase underwent a rigorous testing process. The solution was tested under various sensor network configurations. The test results served as foundation for an evaluation of the developed solution.

1.3 Thesis Outline

The rest of this paper is structured as follows. Chapter 2 presents the related work that influenced the realization of this assignment. Chapter 3 describes the design choices made throughout the development process. The decisions both architectural and implementation specific are reasoned based on the related work in Chapter 2.
In Chapter 4 the detailed implementation process is shown. The report end with Chapter 5, in which the evaluation of the implementation is discussed, before the report is concluded with Chapter 6.

Chapter 2

Related Work

This chapter highlights the related work on which this project was built on. First, the research area Secure Wireless Sensor Networks (SecureWSN) is introduced. SecureWSN is researched at University of Zurich (UZH) and sets the context for this thesis. It contains two main components: Conguration, Management and Data Handling Framework (CoMaDa) and Web-based Mobile Access and Data Handling Framework (WebMaDa), both described in detail. For the implementation part of this assignment CoMaDa is crucial, thus it is described in a more detailed manner than WebMaDa. Second, the current visualization solution in CoMaDa is presented. In the third part of this chapter, possible relational database solutions for this project are highlighted.

The latest important work which lead to this assignment were two contributions to CoMaDa and WebMaDa. The base for the database integration into SecureWSN was done by Claudio Angliker [3], by introducing a new database schema into WebMaDa and restructuring CoMaDa in a way, that a sophisticated intra-applicational communication was possible. Further on Tim Strasser integrated an offline method for visualizing sensor data into the CoMaDa [30]. He implemented a session based visualization methods using Google Charts [12]. Both contributions influenced the design decisions done in this assignment described in Section 3.

2.1 SecureWSN

One research area of the Communication System Group (CSG) at the Department of Informatics at UZH are Secure Wireless Sensor Networks (SecureWSN) [7]. The main goals in this research area are efficient and secure data transmission in WSNs as well as providing a framework for WSNs. These goals imply a variety of requirements that need to be fulfilled. Two categories of requirements can be recognized. Requirements that fall into the first category are requirements for WSNs itself, e.g. efficient data transmission, encryption and authentication in WSNs and pull mechanisms to retrieve data from WSNs on demand and anywhere. The second category of requirements focuses on the management of WSNs, e.g. extensible management frameworks, mobile access to WSNs (see Subsection 2.1.1 and 2.1.2), fine-grained access privileges, secure mobile access and informative visualization of the WSN sensor data.

Many of those requirements were met by numerous contributions over time. Each contribution fulfills some requirements while simultaneously generating new requirements and opportunities to improve the SecureWSN area. The requirement of a database solution for the previously contributed visualization solution built the context of this thesis.
The term *SecureWSN framework* refers to all software components developed by the research group. SecureWSN framework mainly consists of CoMaDa and WebMaDa. The current high level architecture of CoMaDa 1.1 and WebMaDa 1.1 is shown in Figure 2.1.

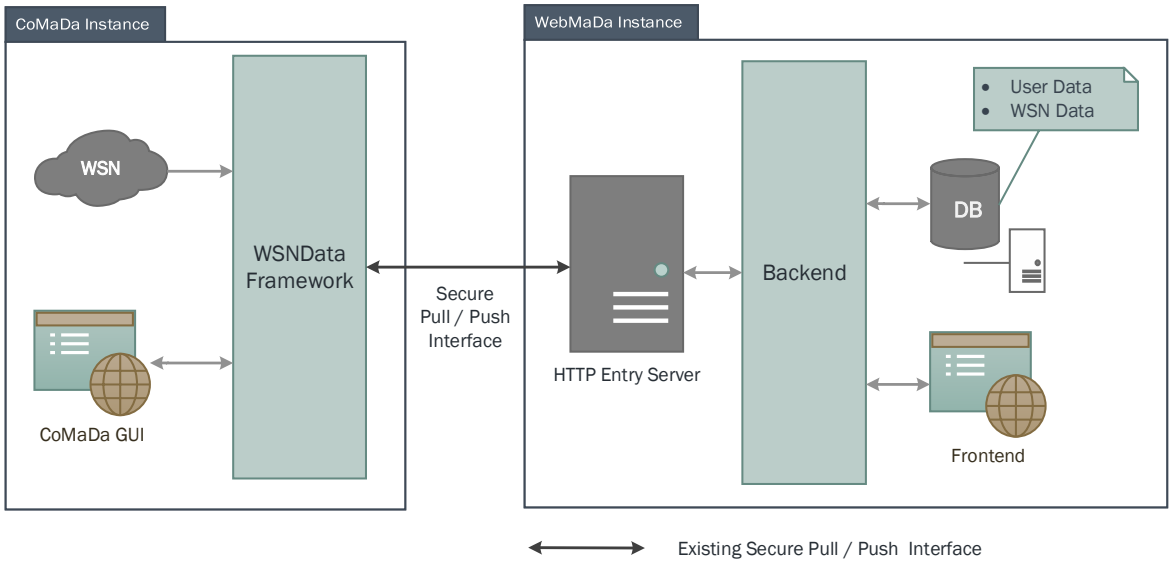

Figure 2.1: Current architecture of CoMaDa 1.1 and WebMaDa 1.1

A CoMaDa instance, thoroughly introduced in Subsection 2.1.1, contains a WSN represented by a cloud symbol, the WsnDataFramework and the graphical user interface (GUI) to manage the sensor nodes and presents the WSN data to the user. A WSN is a collection of sensor nodes that communicate with a base node. The base node is connected over USB to the machine running the WsnDataFramework application. The WsnDataFramework contains two parts; the server backend and the GUI frontend. In Figure 2.1, the CoMaDa GUI is shown separately from the WsnDataFramework to indicate the semantical distinction between the CoMaDa frontend and the backend.
WebMaDa (Subsection 2.1.2) is an extension to CoMaDa, it is a web application and provides mobile access to a WSN as well as a sophisticated user management. As shown in Figure 2.1, CoMaDa and WebMaDa are connected over two secure two-way authenticated interfaces; a pull interface to actively pull data from CoMaDa and an upload interface to receive data from CoMaDa. WebMaDa has a single access point through an entry HTTP server, from there a backend application serves a Bootstrap [5] built frontend and maintains a database to store user data and WSN data from connected CoMaDa's. Both frameworks are described more detailed in the following subsections.

2.1.1 CoMaDa

The CoMaDa framework [11]provides a GUI that offers all needed functionality for creating and managing a WSN in a single standalone application. In the source code, the CoMaDa application is called WsnDataFramework. This is also the term used further on in this thesis for refences to the source code. The WsnDataFramework is implemented as a modularized Java application in a client-server architecture. The server part can be seen as an abstraction layer of a WSN. The server uses different communication drivers to communicate with various sensor node types over a stack of specialized protocols, e.g. TinyIPFIX [10]. The client part, the frontend, consists of HTML, JavaScript [14] and CSS files that combined build the web-application. The HTTP server on the server side serves the files of the GUI to the user and therefore connects the both parts.

The server side of the WsnDataFramework is completely written in Java. It follows a very modularized implementation approach as shown in Figure 2.2. There is a module `HTTPServer` serving the frontend, a module `Web` that orchestrates the communication with WebMaDa over a secure pull and an upload interface.

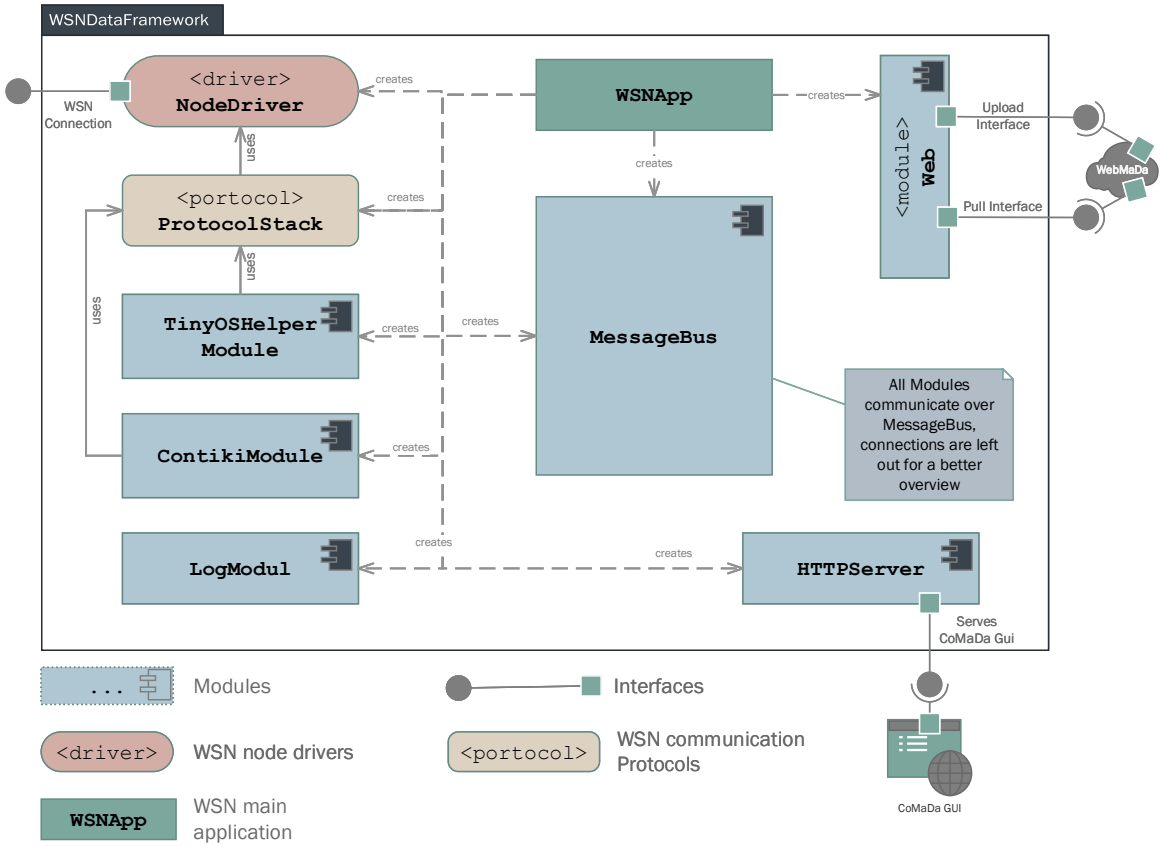

Figure 2.2: Simplified component diagram of the server side from CoMaDa 1.1

All modules are loaded and instantiated into a `WSNApp` instance on the application start. The modules communicate either through thrown events and their respective handler or through a `MessageBus`. Each module can specify custom messages based on an interface and publish them to a message bus instance. The modules are able to subscribe to the message bus for specific message types and handle the incoming messages. It is possible that more than one module subscribes for a message as the name 'message bus' already implies.

The `WSNApp` uses a set of exchangeable drivers and protocols (indicated in the figure with `<driver>` and `<protocol>`) to communicate with the actual WSN. The used driver and the protocols have to be specified in the code before compilation to adapt the application to the used sensor type. Currently, two operating systems are used for the sensor nodes, one is TinyOS [33] the other is Contiki [8]. Both operating systems are represented in the WsnDataFramework as modules (`TinyOSHelperModule` and `ContikiModule`) and present an abstraction layer over the used protocols and drivers for other modules keen on interacting with the WSN. At the time this report was written, only the TinyOS module was implemented in a functional way. Important further on in the thesis is the `LogModule`, it stores incoming WSN data as ordered text files. Currently this is the only way CoMaDa stores the WSN data.
Other, in this context unimportant or unused modules exist in the WsnDataFramework, they were left out of the figure to simplify the visualization. Regarding the figure, it has to be remarked that even if the overall architecture seems elegantly decoupled and easy understandable, there are lots of intermodular dependencies in the framework that complicate the understanding heavily. Therefore, the figure omits the visualization of dependencies in a reasonable way to simplify the diagram. A correct treatment of all dependencies would make the resulting visualization useless.

The structure of the frontend can be seen in Figure 2.3, essentially all frontend code is located in the `html` folder of the WsnDataFramework. The nodes in the figure represent the project folder structure. The 'index' node in the diagram represents the entry point of the GUI with its according CSS, JavaScript and HTML files for each subpage. The node 'widgets' contains reusable components e.g., the grid structure of the GUI pages. Otherwise the widgets folder mainly contains the implementation of different WSN visualizations. The graphical visualization widget in the node 'charts' were written as AngularJS directives and described more detailed in [30] as well as in Subsection 2.1.3. The integration of the database solution developed in this work will primarily happen in the JavaScript file `singelNodeWidgets.js` of the chart widget. The 'topology' node contains the implementation of the visualization of the current node configuration, 'protocols' contains the implementation of the raw data package visualization. The other unmentioned nodes as 'help' or 'pdfs' contain static files used in various pages.

The current structure of the WsnDataFramework was further analyzed to find problems and design flaws introduced through the huge number of different independent contributions. The results of this analysis as well as possible solutions are presented in Section 5.1.

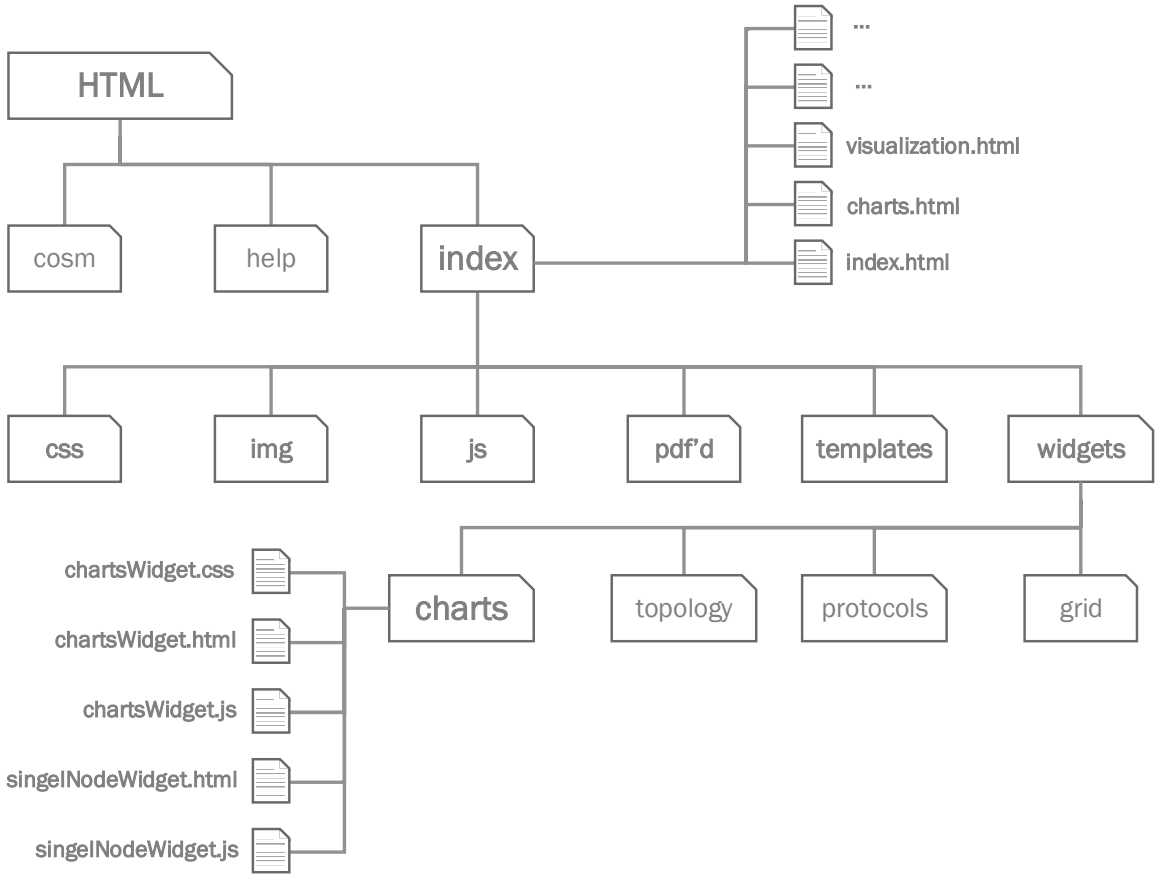

Figure 2.3: Code structure of the client side from CoMaDa 1.1

2.1.2 WebMaDa

WebMaDa extends CoMaDa and provides remote access to registered WSNs. WebMaDa was introduced 2014 at the UZH to close the mobility gap of CoMaDa [25].The WebMaDa GUI is built upon the Bootstrap framework [5] and allows to monitor WSNs devices, as well as actively pull current data from the WSN sensors. WebMaDa has a fine-grained privilege and access management where the owner of a WSN can grant specific access rights to other users of a WebMaDa.
Figure 2.1 at the beginning of this section presents a course-grained overview of the current architecture of WebMaDa. It can be seen that all traffic from a CoMaDa enters through a HTTP server as a single access point, the so-called entry server. Incoming traffic is redirected from the entry server to the PHP based upload interface. A Tomcat server serves the WebSocket endpoints and runs a Java applet for the pull interface. The backend of WebMaDa itself is written in PHP using Ajax to execute the PHP scripts asynchronously. The frontend consists of the Bootstrap web application which is served by an Apache server.
The backend stores the WebMaDa user data and the WSN data in a MySQL database. The MySQL database access is handled over stored procedures to prevent SQL injection and a set of database users with different access rights depending on the executed actions

to maximize database security. The stored procedures are again called by PHP scripts. The database can only be accessed locally over the pull and push interface, which means that no external access outside of the physical machine is possible.

WebMaDa in this state was mostly developed by Claudio Angliker in his master thesis [3] where he introduced the current database design. The database is both concerned with the user management of all WebMaDa users as well as with the WSN data from each attached CoMaDa system. For the work in this thesis, the handling of the WSN data in WebMaDa, specially the database scheme directly influences the design decisions of Section 3. Therefore, the WSN data part of the WebMaDa database is in detail described in the following paragraphs, for information's on the user management part of the database the reader may be referred to Claudio Angliker's thesis [3] where he explains his implementation in Section 4.5.2.

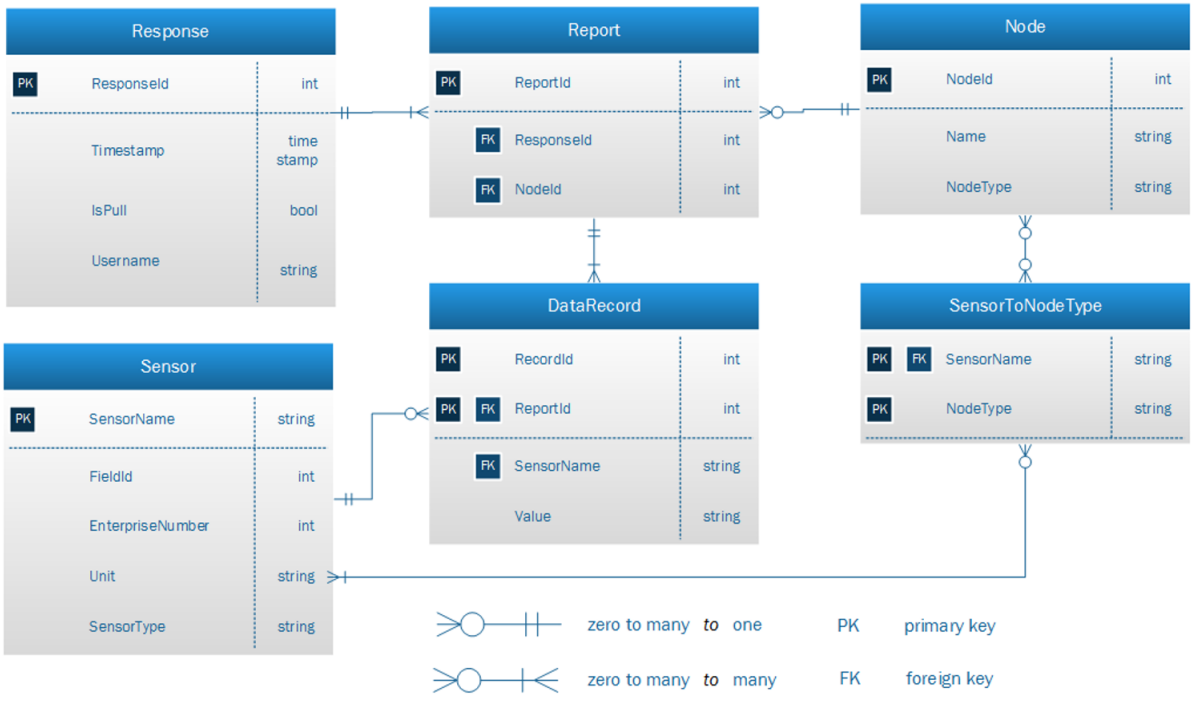

Figure 2.4: WSN Data scheme in Craw's Foot notation [3]

Figure 2.4 presents the database scheme for the WSN data developed by Claudio Angliker in craw-foot notation. Each connected CoMaDa has the six presented tables prefixed with the ID of the WSN and, therefore, guarantees full independence between the stored CoMaDa's. The tables `Node`, `SensorToNodeType`, and `Sensor` store meta information about the sensor nodes, e.g., the unit of a sensor `'Temperature'` on the node with `NodeId` 15. The tables `Response`, `Report`, and `DataRecord` depend on each other and store information on a recorded data value. `Response` stores meta information about the measurement initiator as well as the timestamp of the measurement and if it was a pull or a push measurement. `Response` creates for each entry a unique ID which is used by `Report` as a foreign key. The `Report` table stores the node ID of the sensor node used in the

measurement as well as the response ID. `Report` creates again a unique ID, this ID is used by `DataRecord` as foreign key to build the connection to `Report` and `Response`. The `DataRecord` table stores the measured value together with the sensor name. The foreign key constrains define the insertion order of a new measurement into the database. First, the meta information must be inserted into `Response`, afterwards the `Report` table has to be filled, and finally the actually measured value can be inserted into the `DataRecord` table. The insertion order enforces the data integrity of the database. Partially complete values will not be inserted.

2.1.3 Offline Data Visualization for CoMaDa

The database solution developed throughout this assignment had to be integrated into the current data visualization solution. Tim Strasser introduced a visualization solution based on Google Charts [12]. He used AngularJS [2] directives to build dashboard widgets that show the values of each node sensor (e.g. temperature) as well as aggregated views where the values of a sensor type are shown for all available nodes. Figure 2.5 shows the aggregated view of temperature for a range of nodes. The AngularJS directives

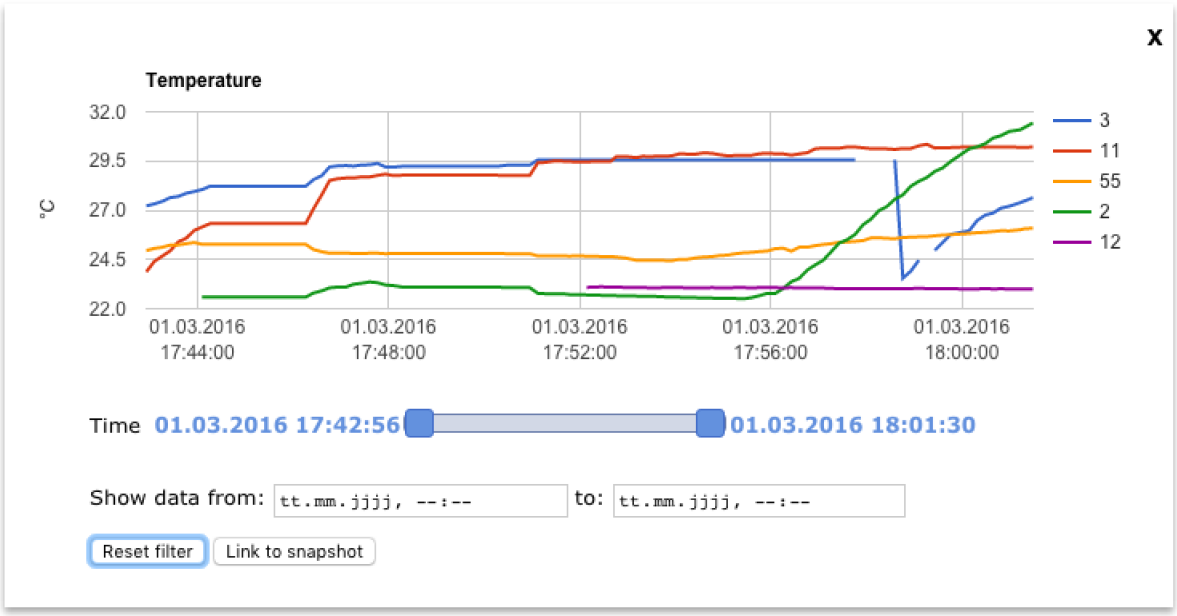

Figure 2.5: Layered chart showing temperature values of several nodes [30]

were written in JavaScript and use HTTP-Get requests to get current information on all available sensor nodes. After the active nodes of the WSN are known to the directives, the current values of the nodes are requested repeatedly, also over HTTP-Get requests. The data of the nodes is then stored in the browsers session storage. After each new data entry Google Charts is triggered to redraw the displayed charts.

The downside of the solution is the session based data storage, it can store incoming data and shows them in the graphs, but only as long as the browser tab is not closed. The integration of the database solution into the directives is shown in Section 4.3.

While investigating this visualization solution, a serious memory leak was discovered. After each redraw the used memory increased consistently. The effects of this memory leak could only be seen after a few hours of displaying the data. After collecting enough data points the cumulated memory allocated on each redraw, reached a point where the web application became unresponsive and eventually crashed. Section 4.3 describes the problems evoked through this leak in more detail.

2.2 Database Solutions

A key factor in this thesis' work is the database solution used. This section presents the most popular open source relational database solutions. The characteristics of each database solution, as well as reasons for and against the usage of a particular solution are highlighted.
Chapter 3 will evaluate the database solutions and reason the final decision on the used database solution based on requirements of the SecureWSN ecosystem.

2.2.1 SQLite

SQLite "is an in-process library that implements a self-contained, serverless, zero configuration, transactional SQL database engine" [1]. It is a library that needs to be embedded into the application, which intends to use the database. SQLite is very popular and integrated in a lot of applications and operating systems e.g. iOS or Android [26]. It runs serverless and without any additional processes involved other than the host application that uses the library, therefore no installation efforts are required. SQLite is single file based, the whole database including all tables, triggers, and views are stored in a single disk file. The database files are completely cross-platform and are not depending on the operating system or its endianness. A database file can simply be copied over to another environment and runs without a problem. The library allows multiple reads at the same time, but only a single write access at any time. In contrast to more complex server based database solutions, SQLite has no user management and does not allow stored procedures. SQLite is favored for applications with embedded databases. The direct file access offers high performance without the need of additional communication channels like sockets or ports. In general SQLite is useful when the application needs to be portable or does not require expansion, e.g. single-user local applications.
Not favored is SQLite in scenarios where multi-user access is needed. First, because SQLite has no user management and, therefore, access privileges on databases and tables are not possible. Second, the single writer limitation of SQLite is a bottleneck, and the application running the SQLite would need to schedule multiple simultaneous write operations accordingly. Further on it is not advised to use SQLite when the database is intended to hold huge amounts of data. [32]

2.2.2 MySQL

MySQL [18] is the most popular open source database management system looking at dynamic web application backends. MySQL has a rich feature set and is easy to start with. Developers have huge amounts of information available on the Internet and can choose from a wide range of third-party applications, tools and integrated libraries that help in the development. To optimize performance MySQL tries not to implement the full SQL standard, but a sophisticated access privilege management as well as the possibility of stored procedures are available. MySQL, unlike SQLite, is a stand-alone database server, and the applications need to talk to MySQL daemon processes to access the database itself.
For the usage of MySQL speak the advanced security features with a fine grained access management as well as a good scalability, MySQL can handle a lot of data and supports a lot of concurrent connections. Through the limitations in the SQL standard implementation MySQL gains performance compared to other database solutions like PostgreSQL. Not favored is MySQL in scenarios where SQL compliance is important. In cases where an integration into a SQL compliant database solution has to be considered, a switch from MySQL may not be easy. Even though MySQL has a rich feature set, it can lack certain features, e.g. a full-text search, which may be important in certain use cases. [32]

2.2.3 PostgreSQL

PostgreSQL [23] is the most advanced, open-source relational database solution and has the main goal of being standards-compliant and extensible. PostgreSQL intends to fully adopt the ANSI/ISO SQL standards. PostgreSQL is programmable and extendible, it allows custom stored procedures not only in SQL but also in other programming languages, e.g., pl/Perl [20] or pl/Python [22]. The stored procedures can be used to simplify complex and repeated database operations, they allow to implement business logic of an application into the database. Again, similar to MySQL, PostgreSQL offers advanced security features and has an access privilege management system. PostgreSQL strongly enforces data integrity through absolute atomic transactions.
Use cases where PostgreSQL may be favorable, include scenarios where data integrity has a high priority, the database has to perform complex business logic or when possible migrations to other database solutions are desired. Specially migrations to proprietary database systems (e.g., Oracle) are easy to handle.
Not favored is PostgreSQL in scenarios where speed is the critical factor, in such cases MySQL may deliver better results. The high complexity of PostgreSQL's configuration for optimal performance may be an over-kill for simple setups.

Chapter 3

Design Decisions

This chapter presents the design choices made in this project. First, architectural considerations on the database integration in the SecureWSN ecosystem are evaluated. Various architecture possibilities are discussed and based on the results, the decision for the implemented solution is justified. Later on, implementation specific choices are motivated based on the findings in Section 2.

3.1 Database Decisions

The main task in this assignment was the implementation of a database solution, which allows the graphical visualization component described in Section 2.1.3 to visualize historical sensor data. Based on this requirement, it had to be evaluated where the database for the sensor data should be located. Figure 3.1 recapitulates the current architectural solution as described in Section 2.1.2. WebMaDa contains a MySQL database, in the figure represented as a data storage bin inside the WebMaDa cloud. The database is integrated into WebMaDa and can only be accessed through the entry HTTP server. It stores user management data and the WSN data from all connected CoMaDa instances, the communication takes place over a secured push and pull interface, which guarantees user validation at any communication step. On the CoMaDa side, all incoming data is logged in files. These given circumstances influenced the decision making in the evaluation process for the location of the database solution. Four possible architectural solutions emerged during the evaluation process:

1. **Single database outside of CoMaDa and WebMaDa**
2. **Single database only inside WebMaDa**
3. **Database for WSN data only in CoMaDa, database only for user management in WebMaDa**
4. **Database for WSN data in CoMaDa, fully replicated database with user management and WSN data in WebMaDa**

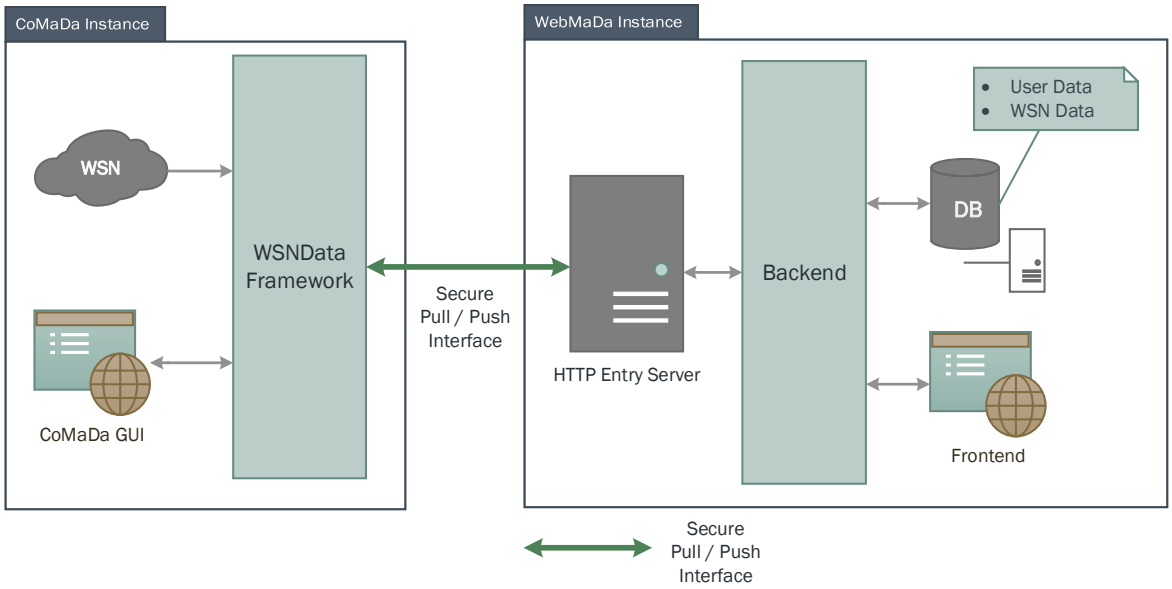

Figure 3.1: Current database architecture of CoMaDa and WebMaDa

In the next subsections, each potential solution will be discussed. For each variant, the upsides and downsides will be shown, especially in regards to security concerns, impact on the current architectural solution as well as impact on further development of the SecureWSN ecosystem.

3.1.1 External Database

This architecture, visualized in Figure 3.2 builds on a single database for WSN sensor data outside of CoMaDa and WebMaDa. A WebMaDa instance would need to store the user data either in the external database or separately in the backend. The WSN data flows from CoMaDa into the database, and both WebMaDa and CoMaDa need to pull the data from the external database for the visualization.

A positive aspect of this database architecture is the single data storage point, external from CoMaDa and WebMaDa. Such an architecture would provide the ability to store data for more than one WebMaDa. A 'Database as a Service' model could be used by a potential distributor of a commercial SecureWSN application.
On the downside, the CoMaDa and WebMaDa frameworks need a new authentication service to access the database that works together with the current WebMaDa access solution. New secure ways to access the database would have to be introduced since the already proven push and pull mechanism between WebMaDa and CoMaDa would be bypassed. Most likely this would mean a dedicated communication protocol and a wrapper around the database. Such an approach would increase the communication traffic massively, since both CoMaDa and WebMaDa need to pull the data from the database for visualization purposes.

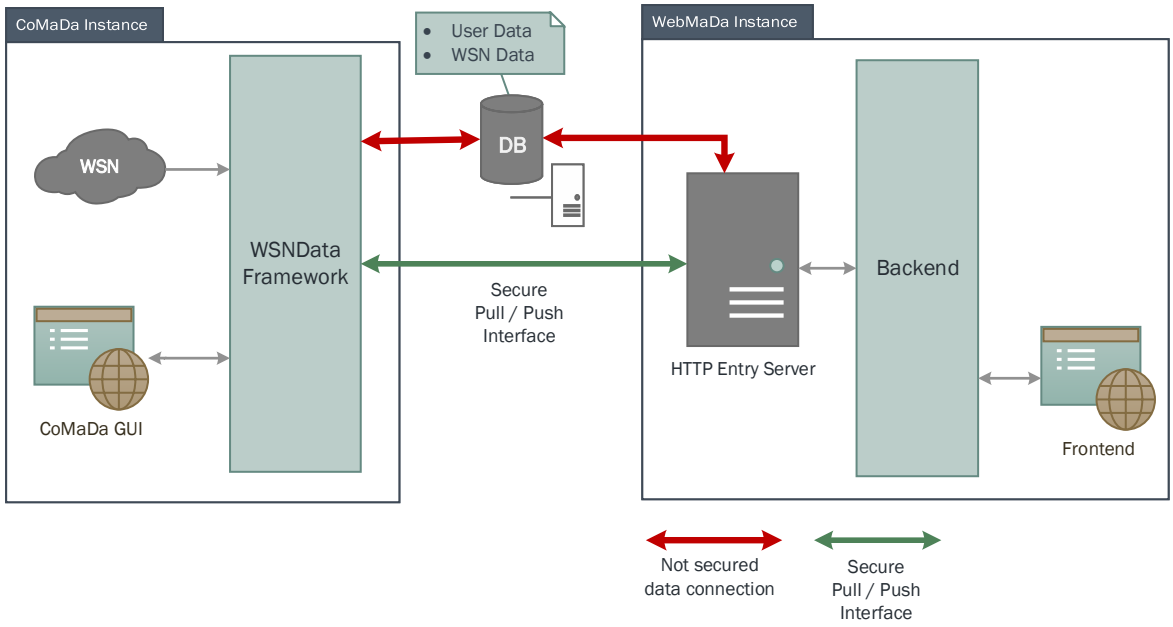

Figure 3.2: External database architecture

The implementation of the external database solution into the current SecureWSN ecosystem requires huge efforts, implementation wise as well as structural. New authentication measures have to be created and the current well established pull and push mechanism would not be important anymore. Considerations to implement this solution only make sense if the whole SecureWSN project would undergo a complete restructuring.

3.1.2 Database in WebMaDa

This architecture is almost identical to the current situation described above in Section 3.1. One database instance inside WebMaDa that contains all data. The data access occurs only over the secure HTTP pull and push interface. The difference to the current implementation is, CoMaDa has the ability to pull data from the database in WebMaDa to satisfy the visualization needs.
In such an architecture, the single data storage point stands out positively. Additionally, the current database could be used without modification and the security concept remains the same.
Downsides would be that the secure pull and push mechanism needs to be extended to allow data retrieval for the data visualization in CoMaDa. The CoMaDa system depends on the availability of WebMaDa to access old WSN data. Like the previous architecture, increased data transfer would be a result of this architecture - the data that already went through CoMaDa has to be retransferred from WebMaDa to CoMaDa just for visualization purposes.

For the implementation of this assignment, the pull and push mechanism would have to be extended and adapted in a way that CoMaDa has the possibility to retrieve data from WebMaDa for visualization purposes. If the changes to the pull and push interface are implemented properly the security of the solution remains unchanged, as the communication goes through the already tested protocol.
For further development of the SecureWSN frameworks, this solution is best suited when the development focuses on a single application approach by combining CoMaDa and WebMaDa. Through the merging of both frameworks, the CoMaDa part could access the database without security concerns or communication performance restrictions. To achieve a complete single application state, an additional database connection layer would be needed in CoMaDa.

3.1.3 Database in CoMaDa

The third database architecture, visualized in Figure 3.3, is a combination of a main database for the WSN sensor data in CoMaDa and a small database for the user management data in WebMaDa. CoMaDa needs a database access layer to access the database directly. Also, the secure push and pull interface needs functionality that allows WebMaDa to pull data from the database for data visualization.

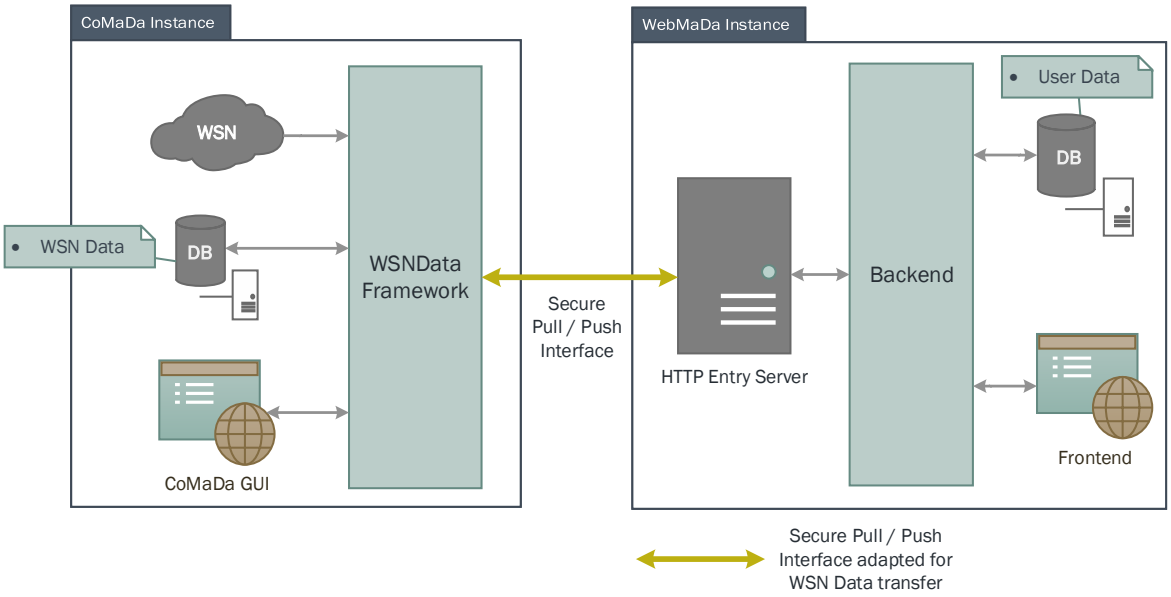

Figure 3.3: Database for WSN data in CoMaDa

Positively stands out that the WSN data is stored locally in CoMaDa, therefore no additional external communication is needed to visualize the data in CoMaDa.
The main challenge with this approach would be, that the secure pull and push mechanism would have to be extended to allow data retrieval for the data visualization in WebMaDa. Again, redundant data transfer would be present - the data that already went into WebMaDa has to be retransferred from CoMaDa to WebMaDa just for data visualization.

The implementation of this approach contains an implementation of a database access layer into CoMaDa as well as an extension of the current pull and push interface for WebMaDa to access the old WSN data. Additionally, the parts of WebMaDa that currently access WSN data need to be rewritten and pull the data from CoMaDa. The workload required to bring this concept completely to work would exceed the timeframe of this assignment.
This approach can be a good option if further development of CoMaDa and WebMaDa tends to go in a clear two component way. This would mean that CoMaDa instances could exist and run without a WebMaDa instance and thus also without Internet connection. Such an implementation also needs a dedicated authentication service for CoMaDa which must be kept in synchronization with WebMaDa.

3.1.4 Mixed Database in CoMaDa and WebMaDa

Presented in Figure 3.4 this architectural solution relies on two databases containing the WSN data. The user management data is exclusively stored in the WebMaDa database. Since the replication happens implicitly through the current pull and push mechanism, no changes to the communication interface have to be done.

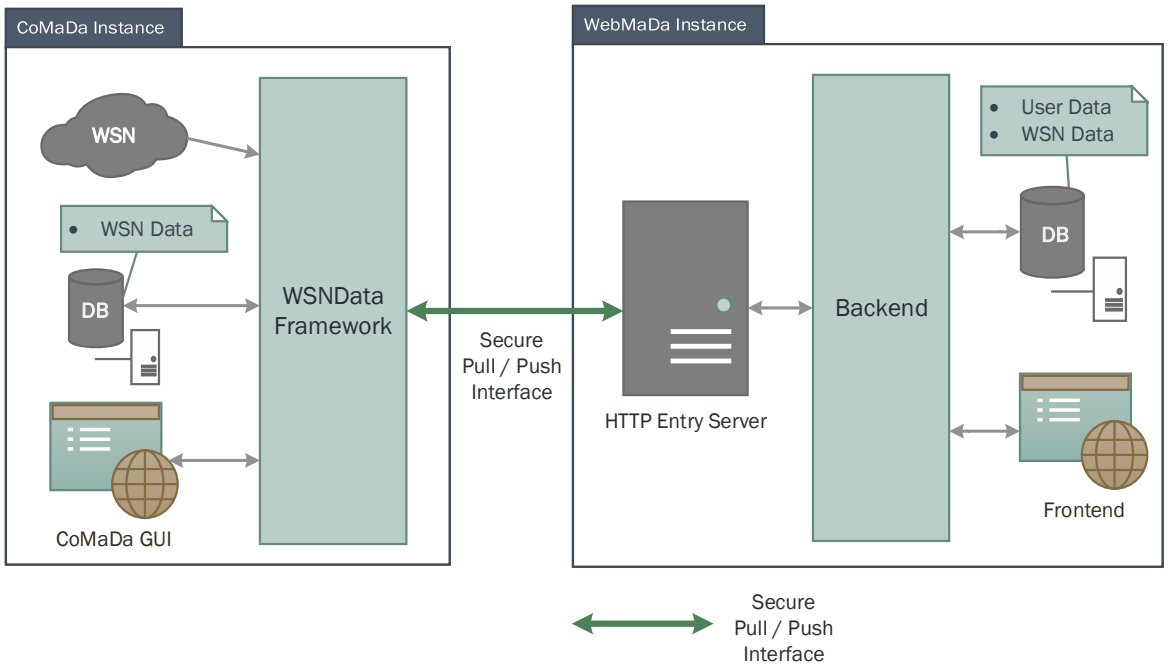

Figure 3.4: Replicated database architecture of CoMaDa and WebMaDa

The following arguments are in favor of this approach: The current database scheme could be used, on the CoMaDa side only the database scheme parts important for the WSN data will be reused. No additional security measures are needed - the database in CoMaDa can only be accessed locally through the WsnDataFramework application and,

thus, no further authentication is needed. The data is locally available for visualization in CoMaDa, no latency through network communication is involved as well as no redundant data transfers. All WSN data goes once into CoMaDa, is there stored in the database, then once into WebMaDa and is again stored in the second database.
On the negative side of this solution could be mentioned that implementation efforts on the WsnDataFramework would be needed to integrate the database into CoMaDa. Further on, it could be said that the complexity of the solution rises due to two databases that both need to be managed.

In terms of further development of the SecureWSN ecosystem this approach is the most flexible solution compared to the other three. If the development steers into a single application approach, e.g. the collapse of CoMaDa and WebMaDa into a single server application with a single frontend, the database integration in CoMaDa can be reused for a single database in the new application due to the identical database scheme.
If the future contributions to CoMaDa and WebMaDa follow a more separated approach, this architectural solution can also be applied. A more separated approach in this context means that a CoMaDa system can be executed independent from the WebMaDa instance e.g. when no internet connection is available.

For the implementation part of this assignment the mixed database approach was chosen. The main reasons for this decision were the high flexibility of the solution for further development on CoMaDa and WebMaDa, also crucial in the decision was the fact that the secure communication between CoMaDa and WebMaDa underlies no changes, which reduces the risk of new security holes in the communication protocol.

3.2 Implementation specific Decisions

After the decision on the architectural structure, implementation specific details regarding the database and the method of integration into the visualization component had to be decided. In this section, first, the database technology selected for the implementation is presented including the motivations that lead to the decision. Later on the decision on the used database scheme is motivated again including the critical decision factors. Afterwards, design considerations regarding the implementation of the database into CoMaDa are presented. Finally, the decisions on how the database solution should interact with the visualization component upon integration into CoMaDa are described.

3.2.1 Database Design

In Section 3.1.4 the architectural decision on the database was made. It was decided to implement a database into CoMaDa and leave the existing one on the WebMaDa side as is. For the implementation phase of this assignment, more technical aspects on the implementation of the database had to be decided. The first decisions that had to be taken were the used database scheme as well as the used database solution. The current database in WebMaDa was implemented with MySQL, other possible technologies were described in Section 2.2.

The decision on the database schema was implicitly given through the fact that the data in CoMaDa should be replicated in WebMaDa. Therefore, the database should be implemented using the same database schema as in WebMaDa (see Figure 2.4) to unify the database approaches.

In the decision process, it was noticed that the used database technology should not be determined once and for all, it must be possible to exchange the used technology without huge implementation efforts. On the WsnDataFramework side, such technology independence should be achieved by using a database abstraction layer that provides a database access interface. The interface specifies functionality required by CoMaDa, later, specialized implementations for a specific database technology implement this interface.

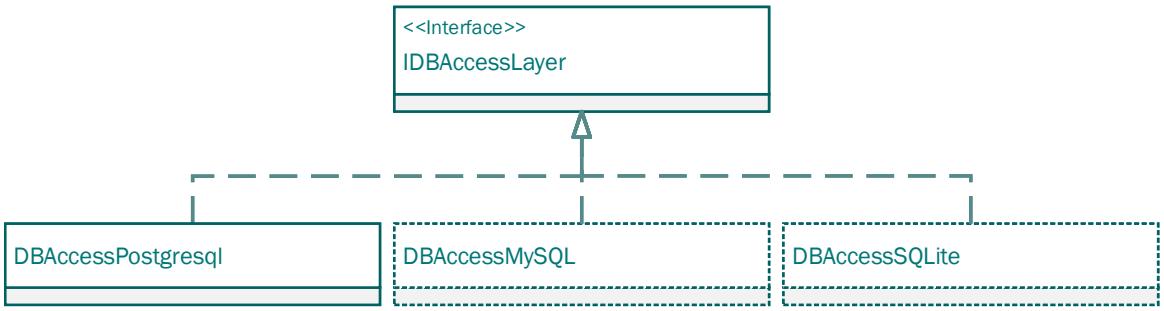

Figure 3.5: Database access layer as interface with possible specializations

Figure 3.5 visualizes the intended Java database access layer interface `IDBAccessLayer` with possible specializations. For the database, PostgreSQL was decided to be used as database solution, thus the `DBAccessLPosgresql` had to be implemented. In the figure, other not jet implemented possibilities were drawn as dashed rectangles. In Section 2.2 it can be seen that PostgreSQL is more or less feature equal to MySQL e.g. it supports the implementation of stored procedures which are heavily used in the database schema in WebMaDa. The translation of the database scheme to PostgreSQL therefore was not a problem. SQLite misses stored procedures functionality, thus the translation of the database scheme would have resulted in a bigger implementation effort. SQLite would be closely integrated into the application and would remove the possibility to use the database when CoMaDa would not be running. A second motivation to use PostgreSQL, was that the database schema would be implemented using a second database technology besides MySQL, further development could revert to either one of the database scheme implementations.

In Section 2.1.1 the component based structure of CoMaDa was shown. To comply with this approach, it was decided to implement a new `DBAccessModule` which acts as database access module. The module is connected to the `MessageBus` and, therefore, allows other modules to connect to the database. Figure 3.6 presents the new module structure including the database module. The module connects to the database over the previously introduced `IDBAccessLayer` and its specializations.

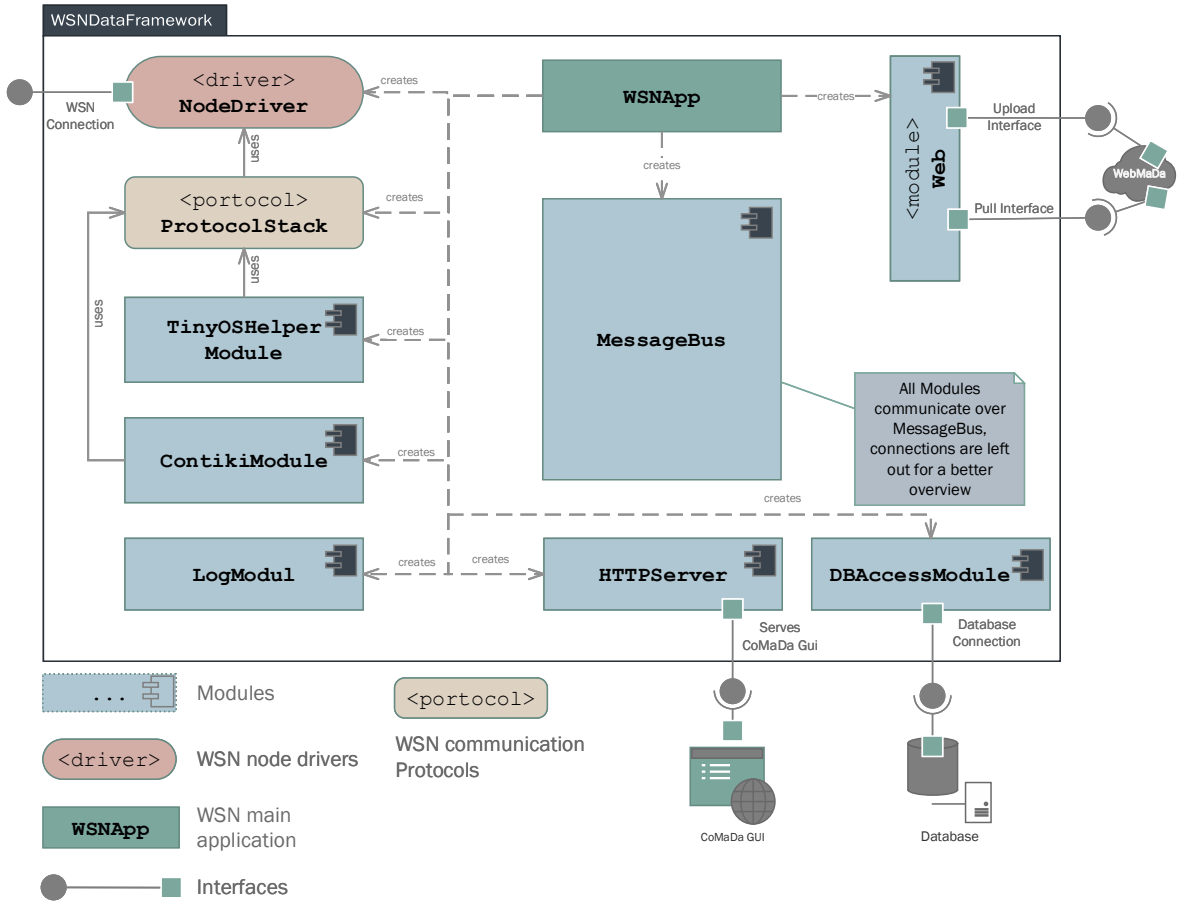

Figure 3.6: New database access module in CoMaDa

3.2.2 Database Integration into the Graphical Visualization Component

In the evaluation of the current graphical visualization component, not only the memory leak broached in Section 2.1.3 was found, but also general performance problems using the Google Charts API for huge data sets were discovered. There are no restrictions on the data size from Google Charts [13], but practically the visualization solution became unusable with more than 3000-4000 data points per chart and nine open charts. Nine Charts equal a WSN with two sensor nodes and three sensors on the node, three charts for each node and three aggregated charts for the three sensors. The problem was not the data size itself, but the recurring redrawing of all charts whenever a new data value has arrived. JavaScript in the browser is at the time single-threaded [16] and, therefore cannot make use of modern multi core central processing units (CPU). The website becomes unresponsive when the browser is not capable to redraw all charts in time. More concrete performance analyzations were made in Section 5.3 where the graphical visualization component was evaluated.
Before the database was integrated into the visualization component, the performance problems could only be seen after a few hours, when enough data points were collected.

With a database linked to the visualization solution, the performance shortcomings were immediately evident. The graphs had to visualize historical data and the website was not usable from the beginning.

To counter the performance issues, two possible solutions emerged. First, the visualization component could be rewritten for the use of D3.js [9]. D3.js is a sophisticated high performance visualization library and would allow to add new data points on the fly without having to redraw the entire graph. Such a solution would not only reduce memory consumption, it would significantly reduce the CPU power needed to draw all charts. The drawback of the D3.js solution would be the immense implementation effort needed to adapt the current solution to the more complex D3.js, therefore, this solution is not practical for this assignment. The second possible solution would be restrictions on the overall drawn amount of data.
At the time this report was written, no practical solution other than restrictions on the displayed data to bypass the performance problems was available. Therefore, it had to be decided to reduce the displayed data in order to keep the visualization solution in working state at acceptable performance.
There are two main 'optimization' points where the amount of displayed data can be reduced. Either over downsampling of old data to reduce the number of data points in a specified interval or over restrictions on the displayed timeframe of the data. Depending of the usage scenario a combination of both may be preferable. A reason for downsampling is also the pointlessness to render more data than available display pixel.
The graphical visualization in CoMaDa is intended to be a 'live component' and is not meant as a data analysis tool for year-old data. It presents the current state of a WSN and therefore the interest in historical data focusses more on the short past.

For the implementation of this assignment it was decided to provide a server sided resampling of the old WSN data, as well as the possibility to restrict the timeframe depending on the use case. Further on, the number of maximal displayed data points in a chart can be configured. All restrictions should be implemented in a configurable way to leave all possibilities open to the user of CoMaDa.

3.3 Summary of Decisions

This section summarizes the decisions on this assignment taken in the previous sections. (1) A database had to be implemented into the CoMaDa infrastructure. The database uses the same database scheme as available in WebMaDa, only the WSN data is stored in CoMaDa. (2) A database access module had to be implemented into the WsnDataFramework that uses an (3) abstracted database access layer to access the database. A specialization for the chosen database technology (4) PostgreSQL of the database access layer had to be implemented. (5) Finally, the database solution had to be integrated into the current visualization component, where restrictions on the amount of displayed data had to be enforced to keep the visualization component usable. (6) Server sided downsampling as well as displayed timeframe restrictions had to be introduced.

Chapter 4

Implementation

This chapter gives detailed insights on the implementation of the work done in this assignment. First, the implementation of the database itself is shown. Second, the integration of the database into the Java backend of CoMaDa and the resulting architecture is presented. Finally, the integration into the graphical visualization component on the frontend side is described.

4.1 Database

This section describes the implementation of the database server as well as the according database scheme. As shown in the design decisions in Section 3 it was chosen to use a PostgreSQL database server [23]. The database scheme from WebMaDa should, based on the decisions, be ported from MySQL to PostgreSQL.
For this assignment the most recent version of PostgreSQL, PostgreSQL 9.6 was chosen, it provides a set of new SQL language features like the `ON CONFLICT` statement, which simplify the porting of the database schema to PostgreSQL significantly. As this version was not yet available over the standard `apt-get` installation command, it had to be installed manually over the PostgreSQL `apt` repository. Listing 1 shows the commands needed to install the PostgreSQL database.

Listing 1: Installation commands of PostgreSQL 9.6 on Ubuntu Linux

```
1 sudo add-apt-repository "deb http://apt.postgresql.org/pub/repos/apt/ xenial-pgdg main"
2 wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
3 sudo apt-get update
4 sudo apt-get install postgresql-9.6
```

The first line adds the repository to the Linux package manager, the second adds the repository validation key to the package manager. The third line updates the list of available packages, and the last line installs the PostgreSQL 9.6 database. The complete installation and configuration process of the database server is thoroughly documented in Appendix A.

PgAdmin III or 4.1 [19] can be used to access and manage PostgreSQL databases, it offers a GUI to interact with the database. Throughout the development, it was discovered that PgAdmin 4.1, the one that officially supports PostgreSQL 9.6, is unstable and hard to use, therefore a combination of both versions is recommended. PgAdmin is able to connect to a database remotely and has not to be located on the same machine as the database. After the installation, a database named `wsndb` was created using PgAdmin, afterwards a database role `wsnadmin` was added to the database server for further encapsulation.

Claudio Angliker delivered a SQL script of the database schema described in Section 2.1.2 for a MySQL database as part of his master thesis [3]. This script called `manual-setup.sql`, contained all information to setup the complete database, including stored procedures. The script was translated part by part to comply with PostgreSQL syntax. On the pure SQL side, e.g. the creation statement of the tables, the differences between MySQL and PostgreSQL were minimal and mostly syntax based. Some type names differed as well as the syntax to describe autoincrementing columns. One big difference was the way of executing the table creation statements. In the WebMaDa database, the tables for the WSN data were created dynamically with a prefix of the current WSN, for the PostgreSQL solution in CoMaDa, one set of tables was created when the complete database was set up. Listing 2 and 3 present the creation statement of the `DataRecord` table in MySQL syntax as well as in PostgreSQL syntax.

Listing 2: MySQL statement to create the `DataRecord` Table

```
SET @s = CONCAT(
'CREATE TABLE ', p_wsnid, '_Datarecord
(
    RecordId INT(128) NOT NULL,
    ReportId INT(10) NOT NULL,
    Value VARCHAR(30) NOT NULL,
    SensorName VARCHAR(50) NOT NULL,
    PRIMARY KEY(RecordId, ReportId),
    FOREIGN KEY (ReportId) REFERENCES ', p_wsnid, '_Report(ReportId),
    FOREIGN KEY(SensorName) REFERENCES ', p_wsnid, '_Sensor(SensorName)
);');
PREPARE stmt FROM @s;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

The creation statement itself is the same in the end, due to the dynamic execution of the MySQL statement based on the `wsnID`, the creation statement had to be constructed appropriately with `CONCAT()` before the statement could be executed.

Listing 3: PostgreSQL statement to create the `DataRecord` Table

```
CREATE TABLE _datarecord (
   recordid integer NOT NULL,
   reportid integer NOT NULL,
   value character varying(30) NOT NULL,
   sensorname character varying(50) NOT NULL,
   CONSTRAINT pk_datarecord PRIMARY KEY (recordid, reportid),
   CONSTRAINT fk_datarecord_report FOREIGN KEY (reportid) REFERENCES _report(reportid),
   CONSTRAINT fk_datarecord_sensor FOREIGN KEY (sensorname) REFERENCES _sensor(sensorname)
);
```

The transition of the stored procedures is shown in Listing 4 and 5. Similar to the table creation statements, the stored procedures perform the task of a value insertion in a similar way. Again, the dynamic structure imposed through the `wsnID` enforces the precreation of the statement in the MySQL example. The PostgreSQL syntax of a stored procedure follows more the classic 'function' syntax known through other programming languages like Java. A MySQL stored procedure is formed through a collection of SQL statements. PostgreSQL allows the use of other languages than pure SQL. In the Listing 5 `plpgsql` [21] was used which extends SQL into a more 'classic' programming language. Other languages that could have been used would be pl/PERL [20] or pl/Python [22].

Listing 4: MySQL statement to create a stored procedure

```
CREATE PROCEDURE Upload_AddResponse(
      IN p_wsnid VARCHAR(30),
      IN p_ispull TINYINT(1),
      IN p_username VARCHAR(50))
BEGIN
SET @p_ispull = p_ispull;
SET @p_username = p_username;

SET @s = CONCAT('INSERT IGNORE INTO ', p_wsnid, '_Response(IsPull,Username) VALUES (?,?);');
PREPARE stmt FROM @s;
EXECUTE stmt USING @p_ispull, @p_username;
DEALLOCATE PREPARE stmt;
SELECT LAST_INSERT_ID() AS LAST_ID;
END //
DELIMITER ;
```

In the end, all WSN related parts of the database scheme were ported for PostgreSQL from the script created by Claudio Angliker. The final database code was again stored in a script that can be executed by PgAdmin or through the command line of PostgreSQL. The next step was the integration of the database into the WsnDataFramework, this implementation step is described in the next section.

Listing 5: PostgreSQL statement to create a stored procedure

```
1  CREATE FUNCTION wsn_add_response(   p_ispull boolean, p_username character varying)
2     RETURNS integer
3     LANGUAGE plpgsql
4     AS
5  $body$
6    DECLARE
7     res integer;
8    BEGIN
9     INSERT INTO _Response(IsPull, Username) VALUES (p_ispull, p_username);
10    SELECT max(responseid) from _response INTO res;
11    RETURN res;
12   END
13 $body$;
```

4.2 CoMaDa Backend Integration

CoMaDa was built in a very modularized manner as described in Section 2.1.1. The implementation of the database integration followed this approach closely. A new module `DBAccessModule` was introduced, this module implemented the predefined module interface `WSNModule`. The database module uses a database handler `BasicDBHandler` that listens on messages and events from other modules and is used to interact with the database. For this interaction a new abstracted database access layer was introduced following the decisions of Section 3. This interface `IDBAccessLayer` is used by the `BasicDBHandler` to connect to the database. As realization of the interface a `DBAccessPostgres` class was introduced, it implements the `IDBAccessLayer` interface and handles all connections to the physical database.
Figure 4.1 presents the class diagram of the new introduced classes described above. For simplicity, only the important methods were listed.

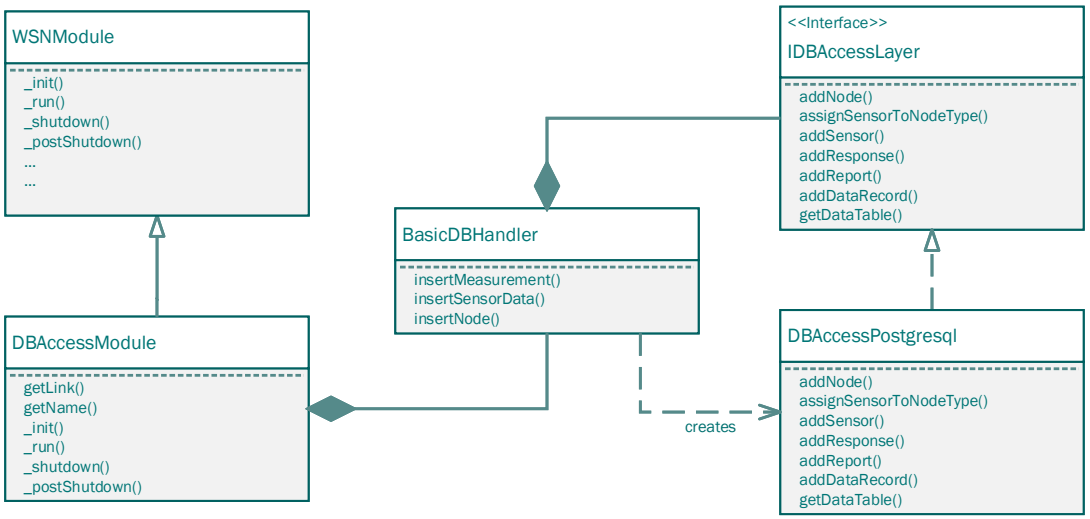

Figure 4.1: Class diagram of the implemented database access module

Listing 6: Database access module for the WsnDataFramework

```
1  public class DBAccessModule extends WSNModule{
2     private BasicDBHandler dbHandler;
3     private String wsnId;
4     private Properties appProperties;
5     private PullIntegration pull;
6
7     public String getLink() {
8        return wsnId;
9     }
10
11    /* overridden methods */
12    @Override
13    public String getName() {
14       return "DB Module";
15    }
16    @Override
17    protected void _init() {
18       _subscribeTo(WSNProtocolPacketProcessedEvent.class, "_event");
19       _subscribeTo(WSNDatastreamChangeEvent.class, "_event");
20       _subscribeTo(WSNTopologyUpdatedEvent.class, "_event");
21       _subscribeTo(WSNNodeUpdatedEvent.class, "_event");
22    }
23    @Inject
24    public DBAccessModule(Properties properties, BasicDBHandler dbHandler){
25       appProperties = properties;
26       wsnId = appProperties.getProperty("wsn.id");
27       this.dbHandler = dbHandler;
28       _setRunning("up and running");
29    }
30    /** list of provided events **/
31    @SuppressWarnings("unchecked")
32    protected Class<? extends Event>[] providedEvents = new Class[]{};
33
34    /** handler, called everytime the module is connected to a new WSN */
35    @Override
36    protected void _run() {
37       try {
38          // wait for module to shutdown
39          this.waitForShutdown();
40       } catch (InterruptedException e) { } finally {}
41    }
42    /** handler that is called everytime the module gets shutdown */
43    protected void _shutdown(){}
44    /*handler that is called everytime the module was shutdown properly*/
45    protected void _postShutdown() {}
46 }
```

Listing 6 presents the straight forward code of the new module class. In the `_init()` method, the module subscribes itself to available events and is therefore able to receive information from other modules in the WsnDataFramework. The constructor `DBAccessModule(Properties properties, BasicDBHandler dbHandler)` sets the module parameters passed by `properties` and the database handler that communicates between

the other modules and the database. The module may provide events (line 34), but this functionality was not needed. The `_run()` method simply waits until the module is interrupted for a shutdown event. Not shown in the listings are the event handler functions, they simply redirect the events to the `BasicDBHandler`.

Listing 7: Insertion function to insert sensor measurements into the database

```
1  /**
2    * This handler initiates the insertion of a sensor measurements
3    * (precisely, it inserts ({@link de.tum.in.net.WSNDataFramework.
4    * Modules.Web.Pull.QueryHandling.Response}) Responses into the database.
5    */
6  @Handler
7  public void insertMeasurements(NewResponseMessage message) {
8     boolean isPull = message.getResponse().isPull();
9     String userName = message.getResponse().getUserName();
10
11    Response response = message.getResponse();
12    response.getReports();
13    // insert response
14    int responseId = dbAccess.addResponse(isPull, userName);
15
16    List<Report> reports = message.getResponse().getReports();
17
18    for(Report rp : reports){
19       int nodeId = rp.getNodeId();
20       // insert report
21       int reportId = dbAccess.addReport(nodeId, responseId);
22
23       List<Record> records = rp.getRecords();
24
25       int recordId = 1;
26       for(Record rc : records){
27          if(rc.getSensorName().equals("Type") ||
28                rc.getSensorName().equals("NodeID")){
29             continue;
30          }
31          String value = rc.getValue();
32          String sensorname = rc.getSensorName();
33          System.out.println("sensorname: " + sensorname);
34          // insert record
35          dbAccess.addDataRecord(recordId, reportId, value, sensorname);
36          recordId++;
37       }
38    }
39 }
```

The database handler `BasicDBHandler` handles message bus messages as well as redirected event calls from the class using the handler. In Listing 7 shows the function that is called when a new sensor measurement message is available on the message bus to inserts the

data into the database. The 'measurement' message contains a `Response` object that is intended for the upload to WebMaDa. In a first step, the user name that initiated the measurement with the information if it was a pull request, is inserted into the database using the `addResponse()` stored procedure. Then, for all reports in the response object, a report is inserted into the database with `addReport()` using the node ID and the response ID generated from the previous insertion statement. Finally, for each report the actually measured data values are inserted into the database using `addDataRecord()`. It can be seen that a datarecord depends on the report which again depends on the response. This dependency of the tables is also enforced through primary- and foreign key constraints in the database schema as shown in Section 3. The other functions in the `BasicDBHandler` follow the same implementation approach and are not further described.
The database access layer interface `IDBAccessLayer` that is used by the `BasicDBHandler` has stubs for all methods used to interact with the database. These methods cover all functionality that is used by the WSNDataFramework at the moment. It can be easily extended for future use cases where additional database interaction may be desired. Java enforces the implementation of all method stubs in the specializations, therefore it is guaranteed that all functionality is implemented.

A specialization for the PostgreSQL database of the `IDBAccessLayer` interface was implemented. The class, called `DBAccessPostgres` implements all methods from the interface. A Java PostgreSQL library [24] was used to connect to the database. Listing 8 presents the method that is used to gather historical data from the database. The function is the most complex one in the class and is too long to be presented on one page, therefore it was split and Listing 9 presents the second part of the function.
In the first listing, the SQL statement that should be executed on the database is described. The statement is dynamically composed depending on the current state of the WSN. For each field of a current node, a `JOIN` operation is added to the statement. The result of the joins is a table containing the timestamp column and a column for each measured value, e.g. temperature, humidity, and voltage of a specific node ID.

The database connection properties as well as restrictions on the requested data can be set by the user in a configuration file. The properties are forwarded to the according modules. In line six to eight from Listing 8 the database connection is established with the configured values. After the SQL statement was composed some parameters had to be set. The node ID was set in line 30, line 36-37 set the start and end interval that should be queried from the database. Decision (6) of the design decisions in Section 3.3 was partly satisfied with the specification of the interval, the downsampling restriction was satisfied later in Listing 9. As seen in line 34 the interval of the requested data was also configurable over a configuration file. In the end, the SQL statement was executed at line 38, and the results put into a hashmap.

The second part of the function to fetch historical data from the database is shown in Listing 9. The listing shows the downsampling restriction decided in Section 3.3. Again, the downsampling was implemented in a configurable way over a configuration file. If the number of returned rows is higher than the specified number of allowed data values in line nine, the data values are downsampled using averaged values.

Listing 8: Data retrieval function for historical WSN data (1)

```
@Override
public ArrayList<Map<String, String>> getDataTable(Properties prop, int nodeId,
                 ArrayList<Map<String, String>> fields) {
  Connection connection = null;
  try{
    connection = connect(   prop.getProperty("db.host"),
                prop.getProperty("db.user"),
                prop.getProperty("db.pw"));
  }catch (SQLException e) {
    System.out.println("DB Connection Failed!");
  }
  PreparedStatement stmnt = null;
  ArrayList<Map<String,String>> res = new ArrayList<>();
  if (connection != null){
    try{
      String stmnt_str = "SELECT r.timestamp as \"Time\"";
      String from_stmnt=" FROM _response r LEFT JOIN _report rp ON rp.responseid = r.responseid ";
      int cnt = 0;
      for (Map<String,String> map : fields){
        cnt++;
        stmnt_str += ", d"+cnt+".value as \"" + map.get("name") + "\"";
        from_stmnt += " LEFT JOIN _datarecord d"+ cnt + " ON d" + cnt +
                ".reportid = rp.reportid and d" + cnt + ".sensorname = '"
                + map.get("name") + "'";
      }
      stmnt_str += from_stmnt;
      stmnt_str += " WHERE rp.nodeid = ? and r.timestamp is not null and r.timestamp > ?
                and  r.timestamp < ? ORDER BY \"Time\";";
      stmnt = connection.prepareStatement(stmnt_str);
      stmnt.setInt(1,nodeId);

      long time = System.currentTimeMillis();
      java.sql.Timestamp ts1 = new java.sql.Timestamp(time -
                props.getProperty("chart.maxdays")*24*3600*1000);
      java.sql.Timestamp ts2 = new java.sql.Timestamp(time);
      stmnt.setTimestamp(2,ts1);
      stmnt.setTimestamp(3,ts2);
      ResultSet rs = stmnt.executeQuery();
      ResultSetMetaData rsmt = rs.getMetaData();
      int columnCount = rsmt.getColumnCount();
      while(rs.next()){
        Map<String,String> resultMap = new HashMap<>();
        for(int i = 1; i <= columnCount; i++){
          if ( i == 1){
            String date = String.valueOf(rs.getTimestamp(i).getTime());
            resultMap.put(rsmt.getColumnName(i).toLowerCase(),date);
          } else {
            resultMap.put(rsmt.getColumnName(i).toLowerCase(),rs.getString(i));
          }
        }
        res.add(resultMap);
      }
    }catch (SQLException e) {
    ...
}
```

Listing 9: Data retrieval function for historical WSN data (2)

```
@Override
public ArrayList<Map<String, String>> getDataTable(Properties prop, int nodeId,
                      ArrayList<Map<String, String>> fields) {

   ...

  ArrayList<Map<String,String>> res_resamepled = new ArrayList<>();
  if (res.size() > 0){
    int size = res.size();
    if (size > props.getProperty("chart.downsampling_nr")){
      int resample_cnt = (size/props.getProperty("chart.downsampling_nr"));
      while (res.size() > 0) {
        Map<String, String> avg = res.remove(0);
        int cnt = 1;
        for (int i = 1; i < resample_cnt && res.size() > 0; i++) {
          Map<String, String> tmp = res.remove(0);
          for (Map.Entry<String,String> e : tmp.entrySet()){
            if(e.getValue() != null){
              if(e.getKey().equals("time")){
                Long old_val = Long.valueOf(avg.get(e.getKey()));
                Long new_val = Long.valueOf(e.getValue());
                Long avg_val = (old_val + new_val) / 2;
                avg.replace(e.getKey(), String.valueOf(avg_val));
              }else{
                Float old_val = Float.valueOf(avg.get(e.getKey()));
                Float new_val = Float.valueOf(e.getValue());
                Float avg_val = (old_val + new_val) / 2;
                avg.replace(e.getKey(), String.valueOf(avg_val));
              }
            }
          }
        }
        res_resamepled.add(avg);
      }
      return res_resamepled;
    }
    else{
      return res;
    }
  }
  return res_resamepled;
}
```

With the implementation of the previously described components, the database integration into the WsnDataFramework was completed. Decisions (1) up to (6) except decision (5) imposed in Section 3.3 were implemented. The dataflow into the database was working, together with the possibility to extract and resample old sensor data. The last implementation step was the integration of the database access into the graphical visualization component and is described in the next section.

4.3 CoMaDa Frontend Integration

After the data was successfully populating the newly integrated database, the integration into the graphical visualization component turned out to be more complicated than previously anticipated. Implementation wise the work was quickly done. One additional HTTP request needed to be handled by the CoMaDa HTTP server to fetch historical data from the database. On the frontend, the AngularJS directives for the visualization had to be adapted in a way that they could request the old data every time the directives were executed for the first time. Both implementation parts are described in detail further on, after the memory leak mentioned in Section 2.1.2 is explained.
After the changes were implemented, the visualization worked as expected in the first few minutes. Old historical data was loaded once at the beginning, from there on new data was added periodically. But after some time, more exactly, after enough data points (400-500) were available, the website with the visualization component started to get unresponsive. The development machine that was used had an Intel i5 CPU and 32GB available random access memory (RAM), performance problems due to hardware limitations could be excluded. Information's from the system performance analyzing tools showed massive memory consumption on the website' process after short execution time. The values rose steadily after each redraw of the graphs in a linear way, and reached from 1.5 GB RAM after a half hour up to 2.6 GB Ram after 1-2 hours until the browser instance crashed. It has to be remarked that at that time, testings were done with a WSN containing two sensor nodes. The maximal number of graphs that could be visualized was nine.
A memory leak was suspected. A first analysis of the code written by Tim Strasser and adapted for the historical data ended without result. Live inspections on the website with the Google Development Tools [6] were also not conclusive. Only the currently active website components were accessible and could be measured, the memory consumption for the active elements was in a low few hundred megabytes which had to be expected given the number of displayed graphs. The precompiled AngularJS JavaScript code was minified before execution, and therefore a meaningful analysis of function calls was not possible, in the debugger function names like `e.Vv()` were the only indications.
Research on performance problems of the Google Charts library indicated memory leaks inside the framework [27] [28]. Most of these leaks were corrected in current API versions, one possible solution mentioned in various comments [27] was the clearing of a chart object before a redraw. This possible solution was tested and ended again without a result. After days of research and debugging sessions it was assumed that the memory accumulates somewhere in a place where it was neither accessible by the debugger because it was already out of scope, nor accessible by the garbage collector. The theory was that the graph objects fall out of scope at a redraw, and are ignored by the garbage collector for unknown reasons.

Throughout further development, various refactoring's of the code were tried to minimize the number of objects that were created and later destroyed. Local variables were lifted into a scope that would only be destroyed once or twice. The final solution that prevented the memory leak was the restructuring of the redrawing function.

Listing 10: JavaScript code of the drawing method for the graphs

```
function drawChart(field) {

   if(ready){
      var fieldDiv = $(iElem).find('#' + field);

      // Create a new chartWrapper
      // The nodeID in the containerId is needed because Google Chart
      // API searches the whole document for the container with this ID
      var options = {
         title: field,
         height: '50%',
         width: '95%',
         vAxis: {
            title: realUnits[field]
         },
         hAxis: {
            format: "dd.MM.yyyy '\n' HH:mm:ss"
         },
         legend: {
            position: 'none'
         }
      };

      var chartWrapper = new google.visualization.ChartWrapper({
         'chartType': 'LineChart',
         'containerId': nodeID + '_' + field.toLowerCase() + '_chart_div',
         'options': options,
         'view': {
            'columns': [0, columns[field]]
         }
      });
      scope.data.charts[field] = chartWrapper;

      // Get the time filter
      filters[field].setContainerId(nodeID + '_' + field.toLowerCase()
               + '_timefilter_div');

      // Draw the dashboard
      var dashboard = new google.visualization.Dashboard(fieldDiv[0]);
      dashboard.bind(filters[field], chartWrapper);
      dashboard.draw(scope.data.dataTable);
   }
}
```

Listing 10 shows the redraw function before the refactoring. It can be seen that the `chartWrapper` object with the chart as well as the dashboard are recreated at each draw. Normally, such a situation would not have negative effects on memory, as soon as the function is finished the memory should automatically be deallocated.
A clear answer why the memory did not get deallocated could not be found, an assump-

tion was stated that the browsers garbage collector had problems detecting the objects through the nested AngularJS directives in the final HTML code. Various Google Chart examples used the technique where the objects are recreated repeatedly [29] but no example could be found using AngularJS directives to discount this theory.
With the memory leak resolved, the memory consumption of nine graphs, each displaying 2500 data points, reached between 200 and 400 MB depending on the machine displaying the website. The memory remained stable over 48 hours of testing. All in all the performance was satisfactory and allowed the use the visualization component without limitations.

Listing 11: Ajax function returning historical WSN data to the frontend

```
/**
* ajax action, returns odldata about the node with the given id
* @param request
* @param response
*
* @author Christian Ott
*/
public void olddataAction(HTTPRequest request, HTTPResponse response) {
   String nodeId = request.arguments.get("id").toString();
   System.out.println(nodeId);

   Node node = this.getServerModule().app().wsn().node(nodeId);
   Map<String,Object> jsonResult = new HashMap<String,Object>();

   // add fields
   ArrayList<Map<String,String>> fields =
                           new ArrayList<Map<String,String>>();
   for (Datum field: node.data()) {
      Map<String,String> f = new HashMap<String,String>();

      f.put("type", field.getType());
      f.put("unit", field.getUnit());
      f.put("name", field.getName());
      fields.add(f);
   }
   jsonResult.put("fields", fields);

   IDBAccessLayer dbAccess = new DBAccessPostgresql();

   jsonResult.put("data", dbAccess.getDataTable(toInt(nodeId),fields));
   System.out.println(jsonResult);
   response.body = JSONValue.toJSONString(jsonResult).getBytes();
}
```

On the server side of CoMaDa an additional HTTP request function was added to the `WSNHTTPIndexController.java` file of the HTTP server. The function was linked to the HTTP-Get request with the URL `/index/oldData?id=NodeID` by the HTTP server module over the naming of the function. Listing 11 presents this function. It is named `olddataAction` therefore the HTTP server listens on `oldData` requests. First the function

extracts the node ID from the HTTP-Get request, then the information's on the node with this ID are collected in line 13. Afterwards all available fields of the node are added to the response. In a last step, the database is queried with help of the database access layer for the historical data of this node. The received data is later added to the response, which is sent back to the browser.

Listing 12: Clientside HTTP-Get request to fetch historical WSN data from the database

```
1  $http.get('/index/oldData?id=' + nodeID).then(function(data) {
2
3     if (data.data != null && data.data != undefined) {
4
5        // Add the time column to the data table that will hold timestamp value
6        scope.data.dataTable.addColumn('datetime', 'Time');
7        var flds = ['time'];
8        for (var key in data.data.fields) {
9           var entry = data.data.fields[key];
10          if (scope.data.blockedFields.indexOf(entry.type) < 0) {
11             // Add the column for this data type to the dataTable
12             scope.data.dataTable.addColumn(chartUnits[entry.type], entry.type);
13             flds.push(entry.name)
14          }
15       }
16
17       for (var key2 in data.data.data) {
18          var entry2 = data.data.data[key2];
19          var keys = Object.keys(entry2);
20          var row = [];
21          for (var k in flds){
22             if ( k == 0){
23                var d =new Date(entry2[k]);
24                row.push(new Date(parseFloat(entry2['time'])));
25
26             }else{
27                row.push(parseFloat(entry2[flds[k].toLowerCase()]));
28             }
29          }
30          scope.data.dataTable.addRow(row)
31       }
32       dataready = true;
33    }
34 });
```

On the frontend side the most important changes were made to `singlenNodeWidget.js`, it contains the AngularJS directive that is used to draw the visualizations of a single node. This directive is further used to get new data values regularly from the WsnDataFramework. The directive was adapted to request the old sensor data from the WsnDataFramework every time it is executed. Listing 12 lists the HTTP-Get snippet used to fetch the old data from the database.

After the data is received, a Google Visualization DataTable is populated with the data. Such a data table can be used as data input for the Google Chart visualizations. Because the HTTP-Get call is made asynchronously, the rest of the directive has to be informed when the data is ready, therefore a flag `dataready` is set to `true` after the function was executed.

With the integration of the database solution into the graphical visualization component completed, the implementation phase of this assignment was brought to an end. All important implementation goals could be completed. The design decisions listed in Section 3.3 were closely followed during the implementation phase. Despite the delay caused by the unexpected problems in the integration of the database into the visualization component, the implementation phase could be completed in time. The result was a flexible database solution in the CoMaDa framework and an enhanced graphical visualization component in the CoMaDa frontend. The next chapter tests and evaluates the results of the implementation phase.

Chapter 5

Evaluation

As described in Section 1.2 the main goal of this assignment was the implementation of a database solution in CoMaDa. The database should be integrated in the existing graphical visualization component to enhance its functionality and enable the visualization of historical sensor data. Before this assignment, the visualization component was session based and could only display real-time sensor data. In the development process, the current state of CoMaDa and WebMaDa should be analyzed as a further task in the assignment. In Section 2.1.1 and 2.1.2 the current state of the frameworks was described in detail.

Section 5.1 describes possible improvements of the frameworks found during the analyzation process of the SecureWSN ecosystem prior to the implementation phase. Theses improvement possibilities may serve as ideas for further contributions to the SecureWSN ecosystem.

Section 5.2 and 5.3 evaluate the results ot the implementation phase in comparison to the previous SecureWSN state. The developed database solution is compared to the existing database solution in WebMaDa. The integration into the visualization component is tested with various WSN configurations and evaluated on performance and scalability.

5.1 Improvement possibilities of the SecureWSN Frameworks

The state of the SecureWSN framework was analyzed in the first phase of this assignment. The familiarization period with the WsnDataFramework and the WebMaDa backend code took longer than estimated, hurdles and information gaps on the mechanics of the SecureWSN had to be overcome. The information's needed to understand the system were distributed over various places. Most information was forwarded over direct verbal communication from previous contributors and the assignment supervisor. Other sources of information were a student wiki [31] where previous work of all students was sparsely

documented, including informative 'How-Tos'. The reports of the previous contributors contained clear information's on their design decisions whereas the technical documentation details were described on an abstracted level. All collected information including the available source code documentation was sufficient for a start of the development phase, albeit a few days of research and experimentation had to be scheduled.
For future contributions, stricter rules on code documentation would, after some time, result in quicker adjustment times. An example what is meant as good documentation would be a structured header on the top of every source code file with the characteristics, dependencies, and internal mechanics of that file, a log of the changes made by each contributor would also be helpful. The naive commenting of the code logic can be helpful but is irrelevant in the understanding most of the time. Throughout the development process, all adapted files were documented this way, if this process can be continued in the next contributions, the respective adjustment times will drop immediately.

More technical remarks on the current state of the frameworks include usability, portability, and configurability of the applications. The WsnDataFramework is heavily coupled with the underlying operating system. Changes to this predefined system, e.g. through distribution of the framework to a newly installed environment, result in path changes throughout the whole source code. Some of them were only detectable at runtime. Therefore, a welcome addition would be the refactoring of the code to set path variables as well as other configuration parameters in one or more configuration files. Changes to the configuration could be made without the need of recompilation. The work done in this assignment tried to follow this configuration based way and made all settings configurable.

The structure of the WsnDataFramework grew through the various additions over the past few years. Naturally, every contributor was focused on his implementation part and a few dependencies to other nodules or components did not have a high impact. System wide these 'few' dependencies per contribution accumulated to a chaos of structural dependencies. The JetBrains Intellij development environment [15] allows the creation of class diagrams from source code. In CD of the submission a zoomable pdf image with the current dependency structure was included, since it would be too big to be included in the report. It can be seen that even if the architecture of the modules may seem decoupled and clearly separated, in reality a web of dependencies exists.
The refactoring that needs to be done to decouple the WsnDataFramework and introduce a clean structure is a rather boring work. Therefore, the refactoring has to be added part by part to future contributions as an additional contribution goal. To begin with, a networking abstraction layer could be introduced that encapsulates the communication to the sensor nodes and presents a standardized interface ready to be used by the other modules. A lot dependencies to current protocols, drivers, and WSN abstraction objects could be resolved.

On the WebMaDa side the code documentation is similarly sparse. The backend contains a set of PHP scripts that originate from two contributions, some of the scripts are currently unused. The decentralized character of WebMaDa, including the database, a HTTP entry server, an Apache server for the frontend, a Tomcat served java servlet for the upload interface, and the PHP scripts as 'glue' between the components, the understanding of the structure involves some efforts. Since the WebMaDa framework was not the core of this assignment, a deep analyzation was not performed. Generally, it can be said, a better

documentation of the code together with a technical file describing the whole WebMaDa architecture including which files are currently in use and which are unused legacy files, would resolve many understanding issues.

To conclude the state of the SecureWSN frameworks, it can be said that, even though some problems or downsides of the current solutions were found, described, and possible improvements proposed, a running system built from ten to fifteen individual contributions is a major achievement. Some caution has to be taken, that further contributors are able to productively develop on the system in reasonable adjustment time. Therefore, a centralized and controlled documentation process is suggested.

5.2 Database Solution

This section evaluates the implemented database solution with regard to flexibility, security, and scalability. The implemented database solution can be compared to the already existing database in the WebMaDa framework. The database exists as a local standalone server in both cases. The database access is restricted to the applications using the database, no external access is allowed. Both databases have the same database schema for the WSN data, WebMaDa additionally stores user data. The main difference is the used database technology, PostgreSQL was used to implement the database on the CoMaDa side, MySQL was previously used for WebMaDa. Both databases can be installed quickly and are ready to use without huge configuration efforts.
A second database option in the SecureWSN ecosystem was provided through the implementation of the PostgreSQL technology, further implementations on the SecureWSN frameworks have the ability to choose which technology suits their needs best. In the current form the databases could be easily exchanged due to the identical database schema. The linkage of the database to the WsnDataFramework was implemented in using an abstraction layer and therefore provides best possible flexibility for further development. The access layer could be adapted to use a secure communication protocol and communicate with an external database. Or the development could be pushed towards a more self-contained solution using SQLite. SQLite would then be embedded into the WsnDataFramework, no additional database servers would be needed. The interface to access the database over the access layer provides the minimal functionality needed to use the database from within the WsnDataFramework, further contributions may extend the usage of the database. Even in the WsnDataFramework integrated automated database creation and destruction may be possible.
An assessment of the database performance was not necessary in the context of the assignment. The current data flow does not exceed one or two insertions per second and one extraction once in a while. Both database technologies are able handle this workload without problems. Further, more complex usage scenarios, including complex and time consuming SQL queries may profit from a better configured PostgreSQL server for the best possible performance.

To summarize the evaluation of the implemented database solution it can be said that the chosen solution offers additional value due to the use of an alternative database technology. The implemented database functionality from a WsnDataFramework perspective may be minimal, but the flexible way of the implementation compensates this feature shortness. Further contribution may steer the database development in any direction, the currently implemented structure is able to serve as foundation.

5.3 Database Integration

This section evaluates the result of the database integration into the graphical visualization component with regard to functionality, performance, and restrictions. In the evaluation process the finally developed solution is compared to the visualization component before the implementation of this assignment.

The integration process of the database was more complicated than expected. One problem was the memory leak presented in Section 4.3, the other problem was the increased number of data points that had to be drawn from the beginning with the database connected to the visualization component. Before the database was connected the charts started with zero data points and eventual performance problems could only be detected after some time. The memory leak could be eliminated, which increased the performance. Nonetheless with enough collected data in the database there were massive performance problems when too much data had to be visualized. As seen in Section 3, this problem was countered with restrictions on the amount of displayed data.
The finally implemented visualization component provided the same functionality as the original visualization solution with the addition of one big improvement. It was now possible to display historical data from a database. Every time the chart is drawn the first time, the old data is fetched from the database. Figure 5.1 presents a screenshot of the old visualization component.

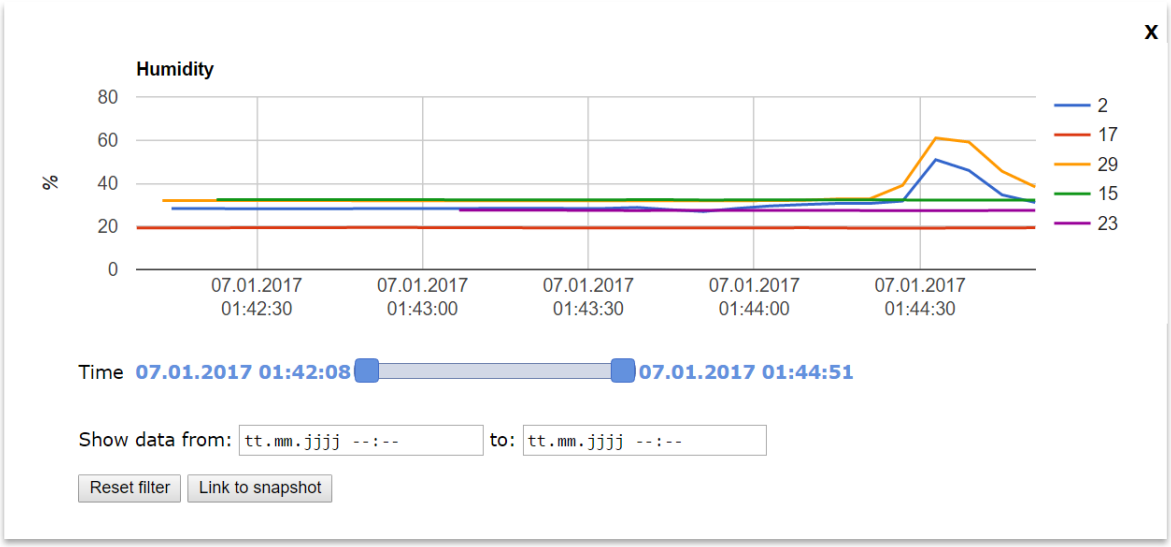

Figure 5.1: Screenshot of the old visualization component

The figure contains around 30 data points for every node, and presents more or less a snapshot of the current humidity situation of various sensor nodes. If the browser tab would be accidently closed, the data would be lost.

Figure 5.2 presents the exact same situation at the same time with the newly developed database integrated visualization solution.

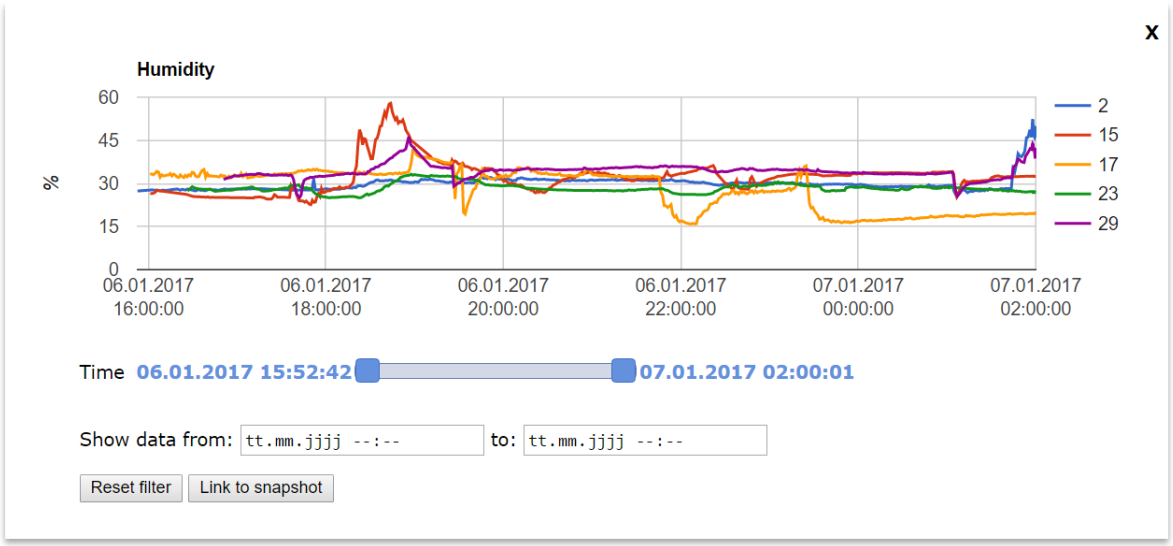

Figure 5.2: Screenshot of the new visualization component

The presented figure has a completely different impact on the user. One can see at first glance how the humidity progressed over 10 hours. The figure displays 500 data points. The number of displayed points was configured before the CoMaDa instance was started. The database contains a significantly higher amount of data points, sampled at a rate of five seconds per data value. This amount of data had to be downsampled to reach an acceptable performance.

The current Google Charts solution is a trade of between the number of information (data points) presented in a single chart and the overall performance of the solution. The number of data points that shall be displayed in each graph can be configured as well as the interval on how far back the data should be fetched from the database. The downsampling of the database data can be configured too, and should be set equal or below the number of displayed data points. The performance of the final visualization solution was measured with various configurations to find an indication on the maximal possible number of data points drawable per graph. The same measurements were made using the old visualization component.

The first performance testing setup was a WSN with five sensor nodes containing each three sensors. Therefore, including the aggregated graphs a maximum of 18 graphs could be displayed at the same time. To test the limits of the visualization solution the refresh

rate of the graphs were set to four seconds. For the measurements with the old visualization an additional time factor was considered since the component starts at zero points, and slowly collects the incoming data. For the developed visualization component, time was not a factor, the number of displayed data points stayed the same and, because the memory leak was eliminated, the resulting memory consumption remained also constant. Table 5.1 presents the result of the measurements with eighteen open graphs, colors indicate the usability of the solution. It can be seen that memory consumption was quickly rising in a linear way despite the low number of data points. The CPU workload settled at around 35% after 10 minutes, it seems that the browser throttles the CPU or it is only able to use one CPU core for the session. After 10 minutes the website got slightly unresponsive, after 20 minutes it was almost unusable, and shortly afterwards the website crashed due to the memory limit in the browser.

Table 5.1: Measurements on 18 open graphs with the old visualization component

Time	Memory	CPU	Data Points
5 Minutes	500 MB	10 %	60
7 Minutes	750 MB	30 %	80
10 Minutes	1000 MB	33 %	100
15 Minutes	1500 MB	35 %	150
20 Minutes	2000 MB	35 %	200

Such a visualization component is unusable for a WSN of this size. With the memory leak a long term surveillance of the sensors even in a smaller WSN fails after a few hours.

A similar measurement was done with the adapted visualization component with the connected database. The results are presented in Table 5.2. The measurement started with 500 data points for a single node visualization. At 500 data points per graph, the website had to visualize 15000 data points all 4 seconds (including the aggregated graphs). At 1000 data points per graph 30000 data point were drawn and first signs of unresponsiveness could be observed. Until 1500 data points the visualization component remained useful, afterwards the website was too unresponsive to be useful. With a reduced refresh rate to 10 seconds, the 2000 data points per graph were usable again. Thus, it can be concluded that the performance limiting factor is the single threaded JavaScript routine. If the CPU is not fast enough to draw all graphs until the next refresh round starts, the website shows signs of unresponsiveness. The memory consumption depends on the number of data points drawn, but was not a critical factor anymore.

Table 5.2: Measurements on 18 open graphs with the new visualization component

Data Points	Memory	CPU
500	200 - 350 MB	5 - 15 %
750	400 - 550 MB	8 -25 %
1000	450 - 600 MB	10 - 30 %
1500	600 - 850 MB	15 -30 %
2000	700 - 1000 MB	35 %

The implemented visualization solution underwent a second test using a smaller WSN of only two sensor nodes. In this test scenario, a high number of 4500 data points had to be displayed. With the smaller WSN, a maximum of 9 graphs could be displayed at the same time. Using 4500 data points for each single node chart, a sum of 54000 data points had to be displayed. Given the case, the data was sampled by the database in 10 minute intervals, a graph could display roughly a month of data using 4500 values.

The performance of the visualization component was measured based on the refresh interval of the graphs. The intention was to find a rough estimation what refresh rates were possible with a high number of data points. The results of the second measurement can be seen in Table 5.3:

Table 5.3: Measurements on 9 open graphs each displaying 4500 data points

Refresh Rate	Memory	Peak CPU
30 s	500 - 600 MB	15 %
20 s	500 - 600 MB	25 %
15 s	500 - 600 MB	30 %
10 s	500 - 600 MB	35 %
5 s	500 - 600 MB	35 %

With 30 seconds as refresh interval, the CPU was most of the time idle at 0% and rose to maximal 15% when the charts were redrawn, the website was responsive all the time. At a redraw rate of 15 seconds, some small stutters could be seen when scrolling through the charts, but only while the charts were drawing. At ten seconds, the time stutters could be seen rose significantly, and at five seconds the website was not usable anymore, and the CPU remained constantly in the thirties.

To conclude the evaluation of the adapted graphical visualization component with the integrated database, it can be stated that the final solution provides a significant enhancement to the previous visualization option. The performance of the visualization component was optimized and can now be used for long term monitoring of WSNs.

Chapter 6

Summary and Conclusions

The SecureWSN framework has been enriched with a flexible database solution to store WSN data on the CoMaDa side in the course of this assignment. The database was implemented in PostgreSQL, a second database technology was introduced besides the already existing MySQL database in WebMaDa. Both databases use the same database scheme to store the WSN data. The access to the new database was restricted to the WsnDataFramework, external access was not allowed to preserve the security model of the SecureWSN frameworks. A flexible implementation solution was chosen for the integration of the database. The resulting flexibility allows the use of different database technologies without huge switchover efforts as well as a simple extension of the developed database functionality in CoMaDa.
The database was integrated into the existing graphical visualization component. The integration enhanced the usability of the visualization solution. With the database connection, it is now possible to view historical sensor data, even if the browser session was closed. To preserve the performance of the solution, restrictions on the amount of displayed data had to be implemented. The restrictions can be configured based on the use case.

Given the goal of a database integration into CoMaDa and the work presented during this thesis, one can draw a first conclusion: The implementation of the database solution was intuitive and could be completed without many challenges. The integration of the database solution into the graphical visualization component however, was intensive and troublesome. Not only an inexplicable memory leak in the visualization solution hindered the development, also performance issues of the used Google Chart technology had to be overcome. The resulting tradeoff, where the number of displayed information is reduced to gain performance is acceptable considering 'live' visualization of sensor data as the main task of the visualization. Future contributions may add a new visualization component, with focus on the analysis of historical data to the CoMaDa ecosystem. For such a statistical component, the 'live'-characteristic falls away and therefore, a different implementation approach on the data exchange between the CoMaDa backend and the frontend could be chosen. An additional statistical visualization component would complement the work developed in this assignment into a complete solution to performantly visualize live data as well as historical data.

Bibliography

[1] About SQLite, URL: `http://www.sqlite.org/about.html`, last visited Jan. 8, 2017.

[2] AngularJS Homepage, URL: `https://angularjs.org/`, last visited Jan. 8, 2017.

[3] C. Anliker, *Secure Pull Request Development for TinyIPFIX in Wireless Sensor Networks*; Master's thesis, Department of Informatics, University of Zurich, Zurich,Switzerland, Nov 2015.

[4] Apache Software Foundation: *Apache HTTP Server Project*; URL: `https://httpd.apache.org/`, last visited Jan. 8, 2017.

[5] The Bootstrap Framework, URL: `http://getbootstrap.com/`, last visited Jan. 8, 2017.

[6] Chrome DevTools, URL: `https://developers.google.com/web/tools/chrome-devtools/?hl=en`, last visited Jan. 8, 2017.

[7] Communication Systems Group (CSG), URL: `http://www.csg.uzh.ch/`, last visited Jan. 8, 2017.

[8] Contiki: *The Open Source OS for the Internet of Thing*; URL: `http://www.contiki-os.org`, last visited Jan. 8, 2017.

[9] D3.js: *Data-Driven Documents* ; URL: `https://d3js.org/`, last visited Jan. 8, 2017.

[10] B. Ertl: *Data Aggregation using TinyIPFIX in Wireless Sensor Networks*; Master's thesis, Department of Informatics, Technische Universität Mänchen (TUM), Aug 2011.

[11] A. Freitag, C. Schmitt, and G. Carle. CoMaDa: *An Adaptive Framework with Graphical Support for Conguration, Management, and Data Handling Tasks for Wireless Sensor Networks*; In Proceedings of the 9th International Conference on Network and Service Management, CSNM, pages 211–218. IEEE, Oct 2013

[12] Google Charts, URL: `https://developers.google.com/chart`, last visited Jan. 8, 2017.

[13] Google Groups - Google Visualization API, URL: https://groups.google.com/forum/#!searchin/google-visualization-api/data$20size/google-visualization-api/5bnCXUms7jo/x-6VJ_7VVC4J, last visited Jan. 8, 2017.

[14] JavaScript, URL: https://www.javascript.com/, last visited Jan. 8, 2017.

[15] JetBrains: *IntelliJ IDEA*; URL: https://www.jetbrains.com/idea/, last visited Jan. 8, 2017.

[16] I. Kantor: *Events and timing in-depth*; URL: http://javascript.info/tutorial/events-and-timing-depth, last visited Jan. 8, 2017.

[17] H. Karl and A. Willig: *Protocols and Architectures for Wireless Sensor Networks*; John Wiley and Sons, Vol. 1, ISBN 0470519231, West Sussex, Great Britain, 2007

[18] MySQL: *The world's most popular open source database* ; URL: https://www.mysql.com/, last visited Jan. 8, 2017.

[19] PgAdmin : *PostgreSQL Tools*; URL: https://www.pgadmin.org/download/, last visited Jan. 8, 2017.

[20] PL/Perl: *PL/Perl - Perl Procedural Language*; URL: https://www.postgresql.org/docs/current/static/plperl.html, last visited Jan. 8, 2017.

[21] PL/pgSQL: *PL/pgSQL - SQL Procedural Language*; URL: https://www.postgresql.org/docs/current/static/plpgsql.html, last visited Jan. 8, 2017.

[22] PL/Python: *PL/Python - Python Procedural Language*; URL: https://www.postgresql.org/docs/current/static/plpython.html, last visited Jan. 8, 2017.

[23] PostgreSQL: *The world's most advanced open source database* ; URL: https://www.postgresql.org/, last visited Jan. 8, 2017.

[24] PostgreSQL JDBC Driver, URL: https://jdbc.postgresql.org/, last visited Jan. 8, 2017.

[25] C. Schmitt, M. Keller, and B. Stiller: *WebMaDa: Web-based Mobile Access and Data Handling Framework for Wireless Sensor Networks (Demo Paper)*; In International Conference on Networked Systems (NetSys), March 2015.

[26] SQLite: *Well-Known Users of SQLite*; URL: https://sqlite.org/famous.html, last visited Jan. 8, 2017.

[27] Stackoverflow: *Google Chart Constant Redrawing Memory Increase*; URL: http://stackoverflow.com/questions/18805964/google-chart-constant-redrawing-memory-increase, last visited Jan. 8, 2017.

[28] Stackoverflow: *Memory leak using google charts with ajax*; URL: http://stackoverflow.com/questions/18686041/memory-leak-using-google-charts-with-ajax?rq=1, last visited Jan. 8, 2017.

[29] Stackoverflow: *Redraw Google Chart after every Ajax call*; URL: `http://stackoverflow.com/questions/18840178/redraw-google-chart-after-every-ajax-call`, last visited Jan. 8, 2017.

[30] T. Strasser, *Offline Method for Graphical Visualization of Sensor Data*; Assignment, University of Zurich, Communication Systems Group, Department of Informatics, Zürich, Swizerland, March 2016.

[31] Student Wiki, URL: `https://wikistudi.corinna-schmitt.de`, last visited Jan. 8, 2017.

[32] O. Tezer: *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems.*; Digital Ocean, 2014, URL: `https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-sy` last visited Jan. 8, 2017.

[33] TinyOS, URL: `http://www.tinyos.net`, last visited Jan. 8, 2017.

[34] Xively, URL: `http://xively.com`, last visited Jan. 8, 2017.

Abbreviations

API	Application Programming Interface
C/S	Client-Server
CoMaDa	Conguration, Management and Data Handling Framework
CPU	Central Processing Unit
CSG	Communication System Group
CSS	Cascading Style Sheet
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of things
P2P	Peer-to-Peer
PHP	PHP: Hypertext Preprocessor
RAM	Random Access Memory
SecureWSN	Secure Wireless Sensor Network
SQL	Structured Query Language
URL	Uniform Resource Locator
USB	Universal Serial Bus
UZH	University of Zurich
WebMaDa	Web-based Mobile Access and Data Handling Framework
WSN	Wireless Sensor Network

List of Figures

2.1 Current architecture of CoMaDa 1.1 and WebMaDa 1.1 4

2.2 Simplified component diagram of the server side from CoMaDa 1.1 5

2.3 Code structure of the client side from CoMaDa 1.1 7

2.4 WSN Data scheme in Craw's Foot notation [3] 8

2.5 Layered chart showing temperature values of several nodes [30] 9

3.1 Current database architecture of CoMaDa and WebMaDa 14

3.2 External database architecture 15

3.3 Database for WSN data in CoMaDa 16

3.4 Replicated database architecture of CoMaDa and WebMaDa 17

3.5 Database access layer as interface with possible specializations 19

3.6 New database access module in CoMaDa 20

4.1 Class diagram of the implemented database access module 26

5.1 Screenshot of the old visualization component 40

5.2 Screenshot of the new visualization component 41

List of Tables

5.1 Measurements on 18 open graphs with the old visualization component . . 42

5.2 Measurements on 18 open graphs with the new visualization component . . 42

5.3 Measurements on 9 open graphs each displaying 4500 data points 43

List of Listings

1 Installation commands of PostgreSQL 9.6 on Ubuntu Linux 23
2 MySQL statement to create the `DataRecord` Table 24
3 PostgreSQL statement to create the `DataRecord` Table 25
4 MySQL statement to create a stored procedure 25
5 PostgreSQL statement to create a stored procedure 26
6 Database access module for the WsnDataFramework 27
7 Insertion function to insert sensor measurements into the database 28
8 Data retrieval function for historical WSN data (1) 30
9 Data retrieval function for historical WSN data (2) 31
10 JavaScript code of the drawing method for the graphs 33
11 Ajax function returning historical WSN data to the frontend 34
12 Clientside HTTP-Get request to fetch historical WSN data from the database 35

Appendix A

Installation Guidelines

This chapter explains how a CoMaDa environment can be extended for the use of a database. Ubuntu Linux in a recent version is assumed as operating system. The configuration possibilities introduced in the development of this assignment are explained as well.

A.1 Database Installation

As described in Section 4.1 a recent PostgreSQL version has to be installed. After following the installation instructions in Listing 1 the database is installed completely and is accessible over `localhost:5432`. For development purposes, a remote access to the database may be desirable. To accomplish this task, two configuration files have to be adapted. In `ostgresql.conf` the entry `listen_addresses` has to be set to `*` or the IP of the remote machine. In `pg_hba.conf` a new entry `host all all 0.0.0.0/0 md5` has to be inserted. The entry allows any database user to access all databases on the PostgreSQL server from any remote address. **Attention! -** revert this changes after development of the database to ensure security!
The Installation of PgAdmin on the development machine is strongly suggested. Since the access for development purposes can occur from a remote machine, the installation of PgAdmin differs depending on the operating system of the host system, therefore the installation instructions from [] may be followed.

After the installation of the database server, a new database user has to be created. After PgAdmin successfully connected to the database server, a new user can be added by right-clicking on `'Login Roles' -> 'New Login Role..'`. As username should `wsnadmin` be chosen.
With the new database user, a new database can be created. This can be done by right-clicking on `'Databases' -> 'New Database...'`. The name should be `wsndb` and the owner the newly created user `wsnadmin`.
After the database was created, the database scheme for the `wsndb` could be built. The content of the file `wsndb-setup.sql` (delivered on the CD of the submission) can be

executed with PgAdmin. After the execution, the WSN database is ready to use for CoMaDa.

A.2 Configuration Possibilities

The developed solution can be configured in the `config.properties` file, located in the directory `WsnDataFramework/conf`. The database connection can be described using the `db.host`, `db.user` and `db.pw` entries. If the installation process described above was closely followed, only the password set by the user has to be changed.

The developed visualization component can also be configured in the `config.properties` file. The property `chart.maxdays` is used to describe how many days backwards the database should be queried for historical data. The downsampling of the data cam be configured using `chart.downsampling_nr`, it specifies the degree of the downsampling.
The refresh rate of the charts has to be configured in the JavaScript files. For the single node charts, the file `singleNodeWidget.js` in the directory `WsnDataFramework/html/index/widgets/charts`, line 33 has to be adapted. A value of 10000 results in a refresh rate of 10 seconds. For the aggregated charts, the file `chartsWidget.js` in the same directory must be changed.
In these files the maximal number of displayed points can be specified in line 336 respectively line 218. This value should be equal or higher than the number specified for the downsampling.

Appendix B

Contents of the CD

The attached CD contains the following files and directories:

- `thesis.pdf`: PDF of the submission
- `abstract.txt`: Plain text version of the English abstract
- `zusfsg.txt`: Plain text version of the German abstract
- `code`: Directory containing the CoMaDa source code, as well as the files needed for database generation
- `tex`: Directory containing the sources of this report
- `presentation`: Directory containing the final presentation