The Sensitive Office

A demonstrator for sensing the enterprise environment

D.J.A. Rennings
S.A. Kassing

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

THE SENSITIVE OFFICE

A DEMONSTRATOR FOR SENSING THE ENTERPRISE ENVIRONMENT

by

D.J.A. Rennings
S.A. Kassing

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science and Engineering

at the Delft University of Technology,
to be defended publicly on Monday July 6, 2015 at 13:30 PM.

Supervisor:	Prof. dr. ir. G. J. P. M. Houben	
Thesis committee:	Prof. dr. ir. G. J. P. M. Houben,	TU Delft
	Dr. M. A. Larson,	TU Delft
	Drs. R. H. J. Sips,	IBM

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

IBM

PREFACE

A mere two months ago we set our first step on the floors of the IBM Centre for Advanced Studies (IBM CAS), taking the leap into an exciting journey. Unbeknown to the challenges that awaited us, we began our endeavour. Resulting in - in the words of our supervisors from both TU Delft and IBM - "*an aspiring project, with potential to excel*". While facing these challenges, we successfully developed a solution that represents the skills and knowledge acquired in our Bachelor of Science in Computer Science and Engineering at the TU Delft.

The accompanying internship at IBM CAS in Amsterdam enabled us to fully analyze requirements, and allowed us to collaborate with its driven employees and interns. IBM CAS offered their continuous support throughout the process, enabling us to consolidate that the intricate system we were creating, fitted within the wishes and ambitions of IBM CAS.

We would like to specifically thank a set of people that were invaluable in the development process. First, we would like to thank our supervisor at IBM, Robert-Jan Sips, for his day-to-day feedback, support, vision, and pragmatism, but also for offering us the opportunity to take part in the *Inclusive Enterprise* research line of IBM CAS, which is currently pending to be patented. Second, we would like to thank our TU Coach, Geert-Jan Houben, for guiding our process and offering advice, next to getting us in touch with IBM in advance of the project. Third, we we would like to thank Manfred Overmeen from IBM for his technical support during the development of the product.

All the courses followed, projects completed, design patterns studied and development methodologies learned, have converged into the creation of this single piece of software. We proudly present the report of our labor, providing both documentation and evaluation of the design, development process and implementation of the system. We hope that you, the reader, will be as captivated as we are by the potential demonstrated by our solution.

D.J.A. Rennings
S.A. Kassing
Delft, June 2015

SUMMARY

The Centre for Advanced Studies of IBM Amsterdam (IBM CAS) has an ambitious plan to revolutionize the workforce. Within the *Inclusive Enterprise* research line they aim to investigate and map the factors that influence employee well-being. Among these factors, we may find environmental factors that can be coupled to measurable quantities such as temperature, light intensity or humidity. IBM CAS approached the TU Delft to create a demonstrator for a system that can measure these quantities by using IoT devices, and collect the data that is produced by such measurements. Based on the wishes of IBM CAS, requirements at both a system- and a data-level have been composed.

Existing solutions were found to be too specific for the fulfillment of a single goal, still in a prototypical state, licensed commercially nonviable, or targeted at a single platform. Based upon a literature study, investigation of previously mentioned existing solutions and thorough analysis of requirements, a client-server architecture was designed. The system architecture consists out of four components: (1) the sensor nodes, responsible for measuring, (2) the hub nodes, responsible for channeling data between the sensor node and the server, (3) the server, responsible for system configuration and data collection, and (4) the database, responsible for storing and providing access to this data. The constructed architecture is not limited to a single context, and can function as a framework for measurement systems.

The strength of the design lies in its platform-independent nature, and its traceable data model. The system has been implemented on the Arduino and Android platform for the sensor node, Java for the hub node, and IBM Bluemix for the server. It has been thoroughly tested using platform-specific frameworks (e.g. JUnit with EclEmma, Mocha with Istanbul) and analyzed using static code analysis (e.g. Sonarqube, SIG evaluation), making it a product of high quality code. The system has been developed with the explicit intention of being continued upon in the future, to grow from a demonstrator into a commercial product.

Contents

1

INTRODUCTION

Over a period of ten weeks, we developed a system that we named "The Sensitive Office". The system is a demonstrator for sensing several phenomena such as temperature, light-intensity and humidity in the enterprise environment by means of a sensor network, a problem that was encountered by the IBM Centre for Advanced Studies (IBM CAS). It was developed as a bachelor end project for the course TI3806 offered by TU Delft at the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS).

It is interesting to note that this project does not solely offer a software implementation to solve the problem of the client. The provided solution is namely a specific implementation of a framework that was designed. This framework allows the reader to implement a measurement system in other contexts as well. The framework has been implemented, and its implementation has been tested and deployed to show the validity of the framework. The created framework is novel, and is currently pending to be patented.

In this report we provide an overview of both the process of creation of the system, as well as the system itself. The report enables the reader to understand the design and implementation of the system. More importantly, it provides the reasoning behind the design and implementation choices, and vital information about the context in which those choices were made. Furthermore, the system is evaluated based on the criteria that were established at its inception.

First, in chapter 2, the research phase is discussed. It covers the analysis of the problem, discusses existing solutions, establishes the requirements, and gives a sense of the adopted methodology for realizing the system. This enables the reader to understand what was demanded by the client, and what the angle of approach of the development team was.

Second, in chapter 3, the design of the system is explained. The architecture of the various system components, their responsibilities and the design choices that back them are given. Furthermore, a detailed description of the data model adopted throughout the system is provided. This architectural overview of the system allows the reader to obtain a high-level understanding of the system and the data model behind it.

Third, in chapter 4, the implementation for each of the designed system components is elaborated upon. It discusses the reasoning behind the choice of hardware and software dependencies, and more importantly: the design of the software. It zooms into every system component, and, through visualizations, it will explain the platform-specific implementation. This enables the reader to quickly understand the underlying mechanics of the implementation for each component of the system.

Fourth, in chapter 5, the testing methodology and results are discussed. The test plan is introduced, and for each of the three employed testing methods (unit, integration and system testing), the results are given and discussed. This enables the reader to understand the engineering effort that has gone into developing a working, reliable and maintainable system.

Fifth, in chapter 6, the final product is evaluated. This evaluation is done through assessing the fulfillment

of the requirements from the research phase, and the usage of objective code quality tools. This provides the reader with evidence for the validation and verification of the system.

Sixth, in chapter 7, the development process is described. This offers the reader with a view on the context of the product development, such as the phases out of which the development process consisted, a distribution of labor in these phases, and interaction with stakeholders throughout these phases.

Seventh, in chapter 8, the future work for the system is discussed. It addresses all extensions and improvements that could not be (partially or fully) done by the development team due to the limited time and/or resources in the project scope. This chapter will enable future project teams to get a good feeling of how to continue with the development of the system in the best manner.

Finally, in chapter 9, the report is concluded and a final word is given by the development team.

2

RESEARCH

2.1. INTRODUCTION

The first step when tackling a problem is the analysis. The need for a thorough, descriptive analysis of the problem allows the creation of effective solutions that tackle the actual problem, which is often masked due to difficulties caused by transferring, translating and transforming of information across cross-disciplinary boundaries [1]. It is important to note that this analysis is merely an inception, and is subject to change throughout the iterative process that software development requires. Requirements and their importance evolve as the system is implemented, and more information becomes available [2].

This chapter will therefore represent the analysis of the problem and provide a basis for creating the actual system as a solution to the problem. We will convince the reader that the chosen approach to the described problem is optimal, based upon our analysis of the problem and the review of alternative solutions. This chapter was created over the first two weeks of the project and contains the result of literary study, but also represents the outcomes of many discussions that were held with stakeholders. It was revised to be included in this report.

First, in section 2.2 the problem is described, existing solutions to the problem are reviewed, the deliverables are laid out, the stakeholders are analysed, and the strategic placement and impact of the system are predicted. Second, in section 2.3 the design goals, which follow from the stakeholders' desires, are defined, making use of ISO standards. Third, in section 2.4 the requirements are laid out, and linked to their respective stakeholder(s) and matching design goal(s). Fourth, in section 2.5 the constraints that are imposed through the context in which the system will be developed, are accounted for. Finally, in section 2.6 the development methodology is described, including our approach to adopting an agile development process.

2.2. PROBLEM DEFINITION AND ANALYSIS

In order to get a good feeling of the problem, this section is devoted to defining the problem, and describing our analysis of this problem. First, in section 2.2.1, the problem and the relevance of solving the problem are described. This description is a more extensive version of the initial problem definition as provided by IBM in March 2015 (see appendix A). Second, in section 2.2.2, the deliverables of the project are listed, and optional research expansions are outlined. Third, in section 2.2.3, existing solutions are examined, in combination with arguments for these solutions not to be an optimal or a complete solution to the problem. Fourth, in section 2.2.4, the stakeholders of the system are enumerated and elaborated upon. Finally, in section 2.2.5, the strategic placement and the potential impact of the project are clarified.

2.2.1. PROBLEM DESCRIPTION

The widespread up-rise of devices that have sensing, computational, and communicative abilities is changing society as we know it. One facet of this interesting development is the Internet of Things (IoT). The IoT is the concept of utilizing "the pervasive presence around us of a variety of things or objects which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach

common goals" [3]. The availability of technologies such as WiFi allow devices to constantly be in connection with all other devices connected to the Internet. As of today, more than a billion intelligent, connected devices already comprise the IoT [4]. The investment required for systems using IoT devices will thus be significantly lower than systems consisting of dedicated self-deployed devices [5]. The low cost of these systems allow deployment at locations where cost is a pressing factor, which would make a solution also deployable in third world countries. As a result, the Internet of Things will have a disruptive effect on various domains, which can be categorized as (1) Personal and Home; (2) Enterprise; (3) Utilities; and (4) Mobile [6].

Among the aforementioned domains we can find the Enterprise Environment. In this domain it gets increasingly important for companies to sense and personalize the (physical) working environment [7]. This personalization has the potential to increase employee well-being, which is generally known to be a sensitive issue, as employers are mainly concerned with the costs [8] rather than the actual well-being. The usage of effective personalization and sensing of an office environment requires highly dense data, on both a spatial and a temporal dimension. For example, one would need to know how light intensity and temperature change throughout the floor of a building to provide either an appropriate seating advice, or an appropriate modification of the temperature.

A variety of partial solutions to the described problem have already been created by others regarding sensing in the Enterprise domain. Unfortunately, these solutions have proven to not be viable, as will be elaborated upon in section 2.2.3. On the other hand, promising developments have been demonstrated in the domain of Agriculture, where low-cost sensing is allowing for highly dense sensing in remote locations [9]. [redacted] [10] [redacted].

[redacted]. This demonstrator can be used as a partial foundation for the development of a larger system, which measures and maintains the employee's pleasure at work. Such a system would comprise of other components as well, in order to analyze the data, and to take into account other parameters such as emotional well-being. The focus of this project is therefore to create a well-documented and well-programmed basis for sensing environmental factors, that can be extended to become (part of) such a full Enterprise Environment system.

2.2.2. DELIVERABLES

As stated by the customer, IBM Centre for Advanced Studies (CAS) Benelux, the focus of this project is to create a demonstrator system in which sensing technologies will be employed in an enterprise environment. If possible (concerning time), the project could also address one or more research questions. Below, the core system components that must be developed within the scope of the project are described first. Secondly, the optional research questions are defined.

CORE SYSTEM COMPONENTS

The system will consist of four[1] core components:

- **Low-cost Sensing Devices:** sensing devices that enable the measurement of phenomena such as temperature, light intensity, sound, and humidity. These sensing devices merely gather sensor data, and send it to the cloud application.
- **Smartphone App:** a cross-platform application for mobile phone devices that enables the measurement of phenomena [redacted] through the embedded sensors, with consent of the user (i.e. owner of the phone). This application merely gathers sensor data, and sends it to the cloud application.
- **Cloud Application:** the central data gathering and aggregation point. The cloud application gains input from the sensing devices, aggregates the data and outputs a transformed version of this data in order

[1] The Low-cost Sensing Devices and the Smartphone App together form the Wireless Sensor Network, throughout this document, the term will thus refer to both components.

to enable the creation of visualizations. It will therefore function as a central point between the sensing devices and visualizations.

- **Visualization**[2]**:** a visualization component providing a representation of the state of an enterprise environment as sensed by the sensing devices.

These core components translate in general terms for the delivery of a) *an architecture document* containing an analysis of the requirements and a description and argumentation for the most important decisions made in designing the system, and b) the *demonstrator code* of the system and related documentation.

OPTIONAL RESEARCH QUESTIONS

Besides the core system, a research component could be added to the deliverables when core components are sufficiently developed and sufficient time can be allocated for the research. The optional research questions below are described at system level, and concern the optimization of the system's performance. The following optional research questions have been composed:

- Which other phenomena could be measured or derived from embedded sensors in smartphones? And, what would be the additional value of taking these phenomena into account?
- What is the optimal distribution between "fixed" low-cost devices and "mobile" sensors to guarantee output and accuracy while minimizing the amount of fixed sensors needed?
- Is it possible to correlate the highly coarse sensing through smartphones and/or low-cost devices to the measurements as done by the existing Building Management System?
- What would be an optimal setup of such a sensing system: Cloud-Centric or Device Centric (the first meaning a centralized cloud instance calculating derivatives from the sensed parameters, the latter meaning local peer-to-peer networks of mobile and fixed devices, jointly deciding on what the most accurate measurement in a specific "zone" would be)?

2.2.3. EXISTING SOLUTIONS

Similar (sub)problems to ours have been encountered by others. In this section we will describe a number of existing (partial) solutions for the problem introduced in the previous section. Existing solutions were examined to uncover their advantages and shortcomings. It will be made evident that these solutions are not sufficient for the context of our problem. However, lessons learned from these existing solutions for associated problems can be used as an inspiration for our own solution.

BUILDING MANAGEMENT SYSTEMS

Modern buildings often contain a building management system, as is the case at IBM CAS in Amsterdam, which operates PRIVA Building Intelligence[3]. Such management systems are mainly aimed at monitoring its system components (e.g. vents, air conditioning, window blinds), and measuring phenomena it can control (e.g temperature, humidity, light intensity). As these systems are specifically designed to regulate the building itself, the granularity at which measurements are taken is generally broad, and the distribution of sensors for these measurements is often done uniformly among the surface areas of the building. This makes such systems unusable for sensing on a human centric scale, in which measurements aim at measuring the conditions of the workspace of a single person. This precision is needed to tackle the problem described in the previous section. Besides the broad granularity and lack of coverage at the scale of individual workspaces, the sensor networks of existing building management systems are not easily extendable to adopt new devices or sensors, as they are connected through physical connections between fixed hardware components. It is therefore not possible to easily extend the system to measure "new" phenomena, which is desired for possible extensions of the demonstrator in the near future. Third, in older and more primitive buildings (e.g. in developing countries) these management systems do not always exist. Deploying such a system in an existing building would be unfeasible, as significant investment would be needed to realize the system's infrastructure. Concluding, existing building management systems are related to this problem, but are far from directly applicable as an off-the-shelf solution for the sensing component of the system.

[2] This component has been moved to future work due to re-prioritization.
[3] http://www.priva.nl//nl/ (retrieved 1 May 2015)

WIRELESS SENSOR NETWORKS

Recent advances in the field of wireless communications and electronics have enabled the development of low-cost sensor networks [11]. Examples of implementations are plentiful, e.g. [10], [12], and [13]. Because this hardware usually does not possess full programming and computing powers but rather a microcontroller, almost all of the related projects found were written in native code. Their code is not reusable in other contexts (or on other device types) without significant porting activities. However, their design methodology can - if legally allowed - function as a basis to develop a similar, but dynamic implementation for low-cost sensor devices. The up-rise of cheap, re-usable, fully programmable Arduino boards[4] supports this. These sensor devices can function as a ground truth. By doing so they allow the calibration of the measurements produced by non-self deployed IoT devices such as smartphones, which are predicted to be less stable. After deploying such low-cost sensor nodes, steps could be taken to solely take into account the non-self deployed IoT devices, of which the measurement may be stabilized by machine learning, allowing to eventually exclude the low-cost sensor nodes. Concluding, these existing wireless sensor network systems are related to this problem, but are again far from being directly applicable as an off-the-shelf solution for the sensing component of the system.

SENSING THROUGH SMARTPHONES

The desire to make use of the sensing capability of smartphones has been shared by others in the past, and has been coined as Human Centric Sensing (HCS) [14]. Existing HCS systems were examined, including frameworks, platforms, libraries, and apps. Two notable solutions were found, being the Medusa and PRISM systems. Though they are not off-the-shelf solutions, but rather prototypes, they show similarities to parts of the context of the problem.

Medusa A programming framework for crowd-sensing that provides support for humans-in-the-loop to trigger sensing actions or review results, the need for incentives, as well as privacy and security [15]. The Medusa framework seems to be primarily focused on the participatory sensing, relying on owners of the smartphone to take action (e.g. take a photo or video) in order to get relevant data. Something which is initially not needed in our solution, as we need continuous sensing and do not want to disturb the user constantly. "The system may never rely on mandatory interactions and additional workload for employees" [7], thus an Opportunistic Sensing approach is preferable. Moreover, Medusa's server is only implemented natively for Linux Ubuntu, and the client respectively for Android. This does not allow for easy portability to other platforms such as Android, iOS, or Windows. There is also little activity in the development of the product [5], no complete documentation, and no license stated, indicating its status as being merely a prototype.

PRISM A Platform for Remote Sensing using Smartphones, which addresses a number of common challenges faced by applications that are aimed at community sensing [16]. It offers system requirements of such a platform, and identifies key design goals in its architectural design. Though, it is currently solely developed for the Windows Phone operating system, whereas support for other platforms such as Android and iOS is preferable for our solution. Moreover, source code, documentation and a complete product were all unavailable. This again hints to a system being merely a prototype, rather than a useful component for our solution.

The analysis of these smartphone sensing solutions led to four conclusions. (1) Many of the systems we found were too specified into a certain direction of sensing aimed at fulfilling a single goal such as traffic management [17]. (2) The systems are often still in a prototype state, not offering full support and documentation which a maintainable future commercial system would require if a dependency exists. (3) The systems are often licensed under GPL or similar licenses, making it difficult to be incorporated in a commercially viable product. IBM therefore requires full control of the Intellectual Property of the product. In the case of PRISM, the source code is even completely proprietary and unavailable. (4) These systems are often oriented at specific platforms only, requiring heavy porting activities to be transferred to other platforms. Concluding, these smartphone sensing solutions are related to this problem, but are not useful as a (partial) solution to our problem.

[4] See http://www.arduino.cc/ for more information.
[5] https://github.com/USC-NSL/Medusa (retrieved 30 April 2015)

2.2.4. STAKEHOLDER ANALYSIS

The system that has to be developed will function as a demonstrator for sensing in an enterprise environment. Thus, the system will not be a fully deployable commercial system. The scope of the stakeholders that are of direct importance is therefore primarily limited to the members of the team of the Centre of Advanced Studies at IBM that are involved in this project. They are the direct stakeholders of this software project, which have immediate stakes in the outcome of the project for various reasons.
The strategic placement of the project also implies a group of indirect stakeholders. These stakeholders need to be taken into account for the project to be of further value after the realisation of the demonstrator, since this would allow for the opportunity to actually create a real commercial product out of the demonstrator. This group of stakeholders consists of the end-users of the system, regarding both development and actual usage. But, as the system that has to be delivered, will only form a demonstrating foundation for the product that will be developed by and for these stakeholders, we have chosen to mainly focus on the direct stakeholders. In order to also take into account the opinion of both groups of indirect stakeholders, we have explicitly asked the IBM research supervisor to represent the opinion of these stakeholders, as he will lead the development team and is already in contact with potential customers. The current (research) state of the product and the role of the members of the development team within IBM does namely not allow for conversations between the development team and potential future customers. In the following subsections you can find a description of both the direct and indirect stakeholders.

DIRECT STAKEHOLDERS

Robert-Jan Sips (RJS) The IBM research supervisor for this project. His concerns are mostly directed at the applicability of the system, and for it to be a demonstrator of the capabilities of one of IBM's strategic responses to the Internet of Things. That the system functions and can be built upon after the project ends are his main concerns. As previously stated, Robert-Jan will also represent the wishes of both groups of indirect stakeholders.

Uditha Ravindra (UR) CS graduate student at IBM. Her master thesis topic - the influence of environmental variables on the work pleasure - is closely related to the system. She will directly use the dataset that will be outputted by the system that has to be developed in this project. She will therefore be taken into account in discussions concerning the design of the system (input), concerning the way data is aggregated and stored, and of course concerning the visualizations of the system (output).

Manfred Overeem (MO) The IBM technical supervisor for this project. His involvement is mostly related to the hardware that is to be used in the project. He will therefore, as a stakeholder, mostly be part of the discussions concerning the hardware used in this project.

Smartphone App Participants (SAP) The IBM personnel that will install the sensing application on their smartphone. Their primary concerns are privacy and energy consumption of the application. This personnel will not be part of discussions other than feedback sessions related to the usability of the smartphone app.

Development Team Members (DTM) The TU Delft bachelor students (Daan Rennings and Simon Kassing) executing this project. Their aim is to fulfill the wishes of the involved direct stakeholders from IBM and TU Delft, and to consider the values of the indirect stakeholders. Next to this, they value an overall successful execution of the project. Choices regarding requirements are, from the perspective of this stakeholder, therefore mainly based upon the ability of the team members to realize a certain component of the system.

INDIRECT STAKEHOLDERS

Future Project Teams (FPT) In the future, the project may be followed up by another project, executed by a different team. Such a team would generally desire high quality code, and clear documentation.

Future Clients (FC) The future clients of IBM that may potentially use the system after its initial deployment at IBM CAS. These end-users will probably attach most value to usability and installability. Since this is a demonstrator, the clients are not listed as a direct stakeholder. Though, throughout the design process, the eventual use of the system by clients is taken into account, as it would be the end-goal of this project to make such adoptions possible in the (near) future.

2.2.5. STRATEGIC PLACEMENT AND IMPACT

The motivation of IBM to support our bachelor thesis comes from the *Inclusive Enterprise* research line within IBM. The goal of this research line is to revolutionize the work environment through technology. The monitoring and maintaining of employee well-being lies at its core. Specifically, there is a focus on the recent development of the Internet of Things, because of the lack of cost to deploy infrastructure among the widespread technology. In the working context, this translates to a variety of things or objects that are spread over the work floor and interact and cooperate with each other to reach a common goal [3]: increasing the work pleasure. Our project finds itself in a larger group of projects that are designed to explore and demonstrate the possibilities in this research line. This pool of projects contains the Poseidon project, and the projects of MSc students Lie Yen Cheung and Uditha Ravindra, which will be described in the following paragraphs. Alongside and in collaboration with these projects, this research line is aimed at changing the working environment to increase the well-being of employees.

POSEIDON PROJECT

The Poseidon project [9], lead by Robert-Jan Sips, explored the usage of low-cost sensing and routing devices to measure the state of plants. In the context of the work environment, it could help to regulate the level of carbon dioxide in the air by signaling when office plants, the converter of carbon dioxide to oxygen, have certain needs. Indication of bad plant health could also be used to infer a problem in the work environment, in essence the plant would become a natural sensor. This project served as inspiration for this project, which encompasses many more sensing, and offers more functionality, but is still aimed at improving the working environment through the use of sensing.

EMOTIONAL WELL-BEING OF EMPLOYEES

Lie Yen Cheung, MSc graduate intern at IBM CAS Benelux, leads a project concerning the emotional well-being of employees on the work floor, as part of her masters' thesis. Using the datasets from an emotional polling game, she attempts to predict how content employees are. In this project she also makes use of the emotions that can be recognized from the faces of employees during their working hours. Although not yet coupled, the state of employees could be laid next to measurable phenomena such as heat, or light in the office, which this system could measure.

ENVIRONMENTAL INFLUENCE ON THE WORK FLOOR

Uditha Ravindra, MSc graduate intern at IBM CAS Benelux, leads another project for her masters' thesis, concerning the influence of environmental variables on the work pleasure, which in turn influences work productivity. The system we are developing in this project will provide her with the datasets of the measurable phenomena at the work floor. We will therefore closely work together with her.

2.3. DESIGN GOALS

The desires of the stakeholders, described in section 2.2.4, are - from a software development perspective - broadly translatable into categories of design goals such as reliability, security, and efficiency. The desires were uncovered after extensive data gathering through discussions and interviews in which the stakeholders formed the central component. First, in section 2.3.1 the methodology for creating a quality model of design goals is formulated. Second, in section 2.3.2 the quality model is concisely described.

2.3.1. ESTABLISHING A QUALITY MODEL

The design goals described in this section are mainly based upon the ISO standards for a quality model of a software product (respectively ISO/IEC 25010 [18] and its predecessor ISO/IEC 9126-1 [19]). The motivation for not merely taking over these standards is the context-dependence of software (development). As [20] states, "design goals do not apply uniformly to all actors, but reflect the perspectives and interests of stakeholders". With this statement and the ISO standards in mind, we defined our own quality model of design goals by also taking into account the stakeholders and our personal expectations of the problem, the context of this problem and the product. The definitions in this quality model are a combination of the explicit standard (ISO) definitions and our own interpretations of the used terms in our context. All of the design goals that are present in the desires of one or more stakeholders are incorporated into our own quality model of design goals.

2.3.2. QUALITY MODEL OF DESIGN GOALS

Using the methodology explained in 2.3.1, a quality model is formulated. This quality model of design goals has been carefully described in appendix B. To allow the reader to get an impression of the model, a comprehensive overview is presented below. The order of the design goals (functional suitability being of the highest importance, down to security being of the lowest importance) gives a rough view of the importance of each design goal.

Functional Suitability *The ability of the system to satisfy stated or implied needs (from stakeholders).*
(1) Functional completeness, (2) functional correctness, (3) functional appropriateness.

Maintainability *The ability of the system to allow maintenance and modification.*
(1) Modifiability, (2) testability, (3) analyzability, (4) reusability, (5) modularity

Portability *The ability of the system to be ported to other contexts, or to handle a changing context.*
(1) Installability, (2) adaptability

Reliability *The ability of the system to perform functions under specified conditions within a specified time window.*
(1) Maturity, (2) fault tolerance, (3) recoverability, (4) availability

Efficiency *The ability of the system to efficiently use what it has at it (potentially) has to its disposal.*
(1) Resource utilization, (2) time utilization, (3) capacity, (4) expense

Compatibility *The ability of the system to exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware and/or software environment.*
(1) Co-existence, (2) interoperability

Usability *The ability of the system to be used by the specified set of users.*
(1) Operability, (2) usability compliance, (3) user interface aesthetics

Security *The ability of the system to actively or passively defend itself from dangers and threats.*
(1) Confidentiality, (2) integrity, (3) authenticity, (4) accountability, (5) non-repudiation

As previously mentioned, design goals are stakeholder-oriented. Instead of arbitrarily linking stakeholders with abstract design goals, the stakeholders' design goals are concretely expressed through the requirements in section 2.4. The description of importance of each requirement in section 2.4 also enables the reader to deduce the importance of each design goal at a system level.

2.4. REQUIREMENTS ANALYSIS

Extensive information has been collected from the stakeholders that are listed in section 2.2.4, primarily through in-depth interviews and discussions. Previously, in section 2.3, the norms and values of the stakeholders were translated into design goals oriented at software development. This section translates these desires into concrete prioritized requirements for the system. First, the strategy that was handled to gather and formulate the requirements is described in section 2.4.1. Second, the overall requirements that are generally applicable to all parts of the system are described in section 2.4.2. Third, the requirements that are imposed upon the three main components of the system are described: sensor network (section 2.4.3), cloud application (section 2.4.4), and visualization (section 2.4.5). Finally, in section 2.4.6 the expectations and success criteria of the project are discussed.

2.4.1. REQUIREMENT CRITERIA

There is a vast diversity in ways to compose requirements for software projects. Using our own experience and education as a guideline, we formulated our own strategy for gathering the requirements for this project. First, the assignment of importance is described. Second, the objectivity and unambiguity of the requirements is elaborated upon.

ASSIGNING IMPORTANCE

The degree of importance for stakeholders are assigned using the MoSCoW method [21]. We adopted this model for prioritizing the requirements as we are familiar with it and experienced its success in previous software development projects. The MUST ('must-have') requirements are guaranteed to be delivered in the scope of the project. The SHOULD ('should-have') requirements should be delivered, are important, and may be painful to leave out. The COULD ('could-have') requirements are wanted or desired, but could be left out with less impact. The WON'T ('won't-have') requirements, are those of which the project team has agreed upon to not deliver, but may be considered in future work on the project. For each requirement in the tables below, the order of importance is assigned by the usage of one of the words "must", "should", "could" or "won't".

MEASURABILITY

During the definition of the requirements, attention was also given to the measurability of the requirements. It must be objectively and unambiguously clear when requirements are met, and when they are not, similar to a boolean expression. Concerning this characteristic, the requirements thus differ from the design goals stated in section 2.3 from which they originate, which are hardly measurable.

2.4.2. GENERAL REQUIREMENTS

Certain design goals, such as maintainability, can be defined on a system level, rather than only making sense on a specific component. The requirements listed below are applicable to all components of the project.

Nr.	Requirement	Design goals	Stakeholder(s)
G1	All functions in the delivered code *must* be accompanied by a comment explaining its input, behavior, and output.	Modifiability, testability	FPT, DTM
G2	The delivered code *must* be developed according to the test plan (to be) described in section 5.2	Maintainability, Reliability	FPT, DTM

2.4.3. SENSOR NETWORK REQUIREMENTS

The sensor network is dedicated to measuring several phenomena such as temperature, light intensity and humidity. These measurements are performed continuously, and thus continuously produce data, which needs to be sent to the cloud application. The sensor network encompasses both sensor nodes, and - if necessary - supporting infrastructure.

Nr.	Requirement	Design goals	Stakeholder(s)
S1	The sensor network *must* be able to measure temperature	Functional completeness	RJS, UR
S2	The sensor network *should* be able to measure light intensity	Functional completeness	RJS
S3	The sensor network *could* be able to measure the level of sound	Functional completeness	RJS
S4	The sensor network *should* be able to measure the level of humidity	Functional completeness	RJS
S5	The sensor network *won't* be able to measure the amount of people in the proximity of sensor nodes	Functional completeness	UR
S6	The sensor network *should* be adaptable to measure other phenomena	Functional completeness, Adaptability	RJS
S7	The sensor network *must* be able to measure the stated phenomena using measurements from sensor nodes with a fixed location (x, y, z)	Functional completeness	RJS, MO, UR
S8	The intervals in which phenomena are measured *should* be adjustable	Functional correctness	RJS, UR
S9	The sensor data *must*, when stored, be labeled with a system-wide synchronized timestamp	Functional correctness	RJS, UR
S10	The sensor network *must* enable new sensor nodes to be deployed within ten minutes	Adaptability	RJS, MO, DTM
S11	The sensor network *must* be able to deliver its sensing data to the cloud application	Functional completeness	RJS
S12	A sensor node *could* communicate with the cloud application while maintaining the confidentiality, integrity, authenticity and accountability of the messages.	Functional completeness, Authenticity, Accountability, Confidentiality, Integrity	RJS, FC, SAP
S13	A sensor node *should* be configured with less than 10 device-specific values.	Installability, adaptability	RJS, MO
S14	The sensor network *should* be able to handle dynamic addition, and removal of sensor nodes.	Adaptability	RJS

LOW-COST SENSING DEVICES SPECIFIC REQUIREMENTS

Requirements specifically aimed at the Low-cost Sensing Devices deliverable.

Nr.	Requirement	Design goals	Stakeholder(s)
L1	The low-cost sensor nodes *should* not make noise that disturbs employees.	Usability compliance	SAP, FC
L2	The low-cost sensor nodes *could* enable the dynamic addition and removal of sensors.	Adaptability	RJS
L3	The sensor network *won't* be able to measure the stated phenomena using measurements from low-cost sensor nodes with a variable location (x, y, z)	Functional appropriateness	RJS, MO, UR
L4	The average price of a readily deployable low-cost sensor node, including supporting infrastructure, *must* be less than $100.	Expense	RJS, MO, FC
L5	The average price of a readily deployable low-cost sensor node, including supporting infrastructure, *should* be less than $80.	Expense	RJS, MO, FC

SMARTPHONE APP SPECIFIC REQUIREMENTS

Requirements specifically aimed at the Smartphone App deliverable.

Nr.	Requirement	Design goals	Stakeholder(s)
A1	When interaction is required, it *should* occur on voluntary basis [7]	Usability compliance	FC, SAP, RJS
A2	When interaction is required, it *should* occur in an engaging context. [7]	Usability compliance	FC, SAP, RJS
A3	The app *should* be easy to use for a non-developer, minimizing the cognitive burden to the user [7].	Operability	SAP
A4	The style of the app *could* match the IBM style.	User interface aesthetics	RJS
A5	The mobile application *should* be well documented so that it can be implemented on different operating systems as well.	Reusability, Installability	RJS
A6	The sensor network *should* be able to measure the stated phenomena using measurements from smartphone devices with a variable location (x, y, z)	Functional appropriateness	RJS, MO, UR

2.4.4. CLOUD APPLICATION REQUIREMENTS

The cloud application will contain a database and run on a central server. This server will receive semi-raw sensor data from the sensor nodes in the network, as they will not send the data they directly gain from their sensors, but in a specific format that will be defined later. The cloud application receives, pre-processes (if necessary), and subsequently stores sensor data in a database. Besides the gathering and storage of data, the cloud application is responsible for interpreting the data, and deriving semantic meaning. It is responsible for making this data available for visualization in a format that allows to visualize it easily.

Nr.	Requirement	Design goals	Stakeholder(s)
C1	The data *must* be stored in with sensor type	Functional completeness	UR, RJS
C2	The data structure *must* have both textual and visual documentation for each entity, and the relations between entities	Modifiability	FPT
C3	The style of the cloud application interface *could* match the IBM style.	User interface aesthetics	RJS, SAP
C4	A basic cloud application interface that allows monitoring of the states of sensor nodes *must* exist	Analyzability, testability	DTM, RJS, MO
C5	The cloud application *must* be able to handle at least ten sensor nodes simultaneously.	Functional completeness	RJS, UR
C6	The cloud application *should* be able to handle 50 sensor nodes simultaneously.	Functional appropriateness	RJS, UR
C7	The cloud application *should* be scalable, and thus scale with the addition of new resources.	Functional appropriateness	RJS
C8	Every endpoint of the access API *must* be completely documented.	Modifiability	RJS, FPT, DTM
C9	Sensor data *must* be traceable to a specific sensor node.	Accountability, Non-repudiation	RJS, DTM, UR

2.4.5. VISUALIZATION REQUIREMENTS

The visualization will make the sensor data accessible through a visual device such as a laptop or tablet. It pulls data in a presentable format from the cloud application, and uses the pulled data to visually represent the state of the system, either textual or graphical.

Nr.	Requirement	Design goals	Stakeholder(s)
V1	A simple statistics visualization *must* be developed	Functional completeness	RJS
V2	A graphical visualization *could* be developed	Functional completeness	RJS
V3	The style of the visualization *could* match the IBM style.	User interface aesthetics	RJS, SAP, FC
V4	Each visualization *must* (be able to) retrieve the visualization data from the server via the Internet.	Functional completeness	RJS, MO

2.4.6. EXPECTATIONS AND SUCCESS CRITERIA

Before starting a software project, it is important to estimate when the project is completed successfully for the specified period of time. Such estimations should be made in accordance with the client. Together with the direct stakeholders, we therefore defined the project to be successful when a basic static sensing system for the office environment is created that forms a maintainable basis for future development. It is important to note that all of the criteria must be fulfilled under the context constraints listed in section 2.5. In terms of requirements, we can translate this definition of success regarding the four categories of the MoSCoW method. That way, the minimal success criterion was relatively simple to define: all MUST requirements

need to be met within the time frame of the project (ten weeks from the start, so within eight weeks from the publication of this report). Furthermore, additional SHOULD or COULD requirements that are implemented, will add to its success. In our expectation, hurdles will be found during the development cycles, resulting in the need to add, remove, redefine and/or re-prioritize requirements, a phenomenon that is commonly known as "software evolution" [2]. We will therefore aim at fulfilling all of the 'must haves' in accordance with the context constraints and fulfilling as much of the other requirements (should requirements prior to could requirements) as possible, while keeping the evolution and continuation of our software product into account.

2.5. CONTEXT CONSTRAINTS

This section is devoted to describing, and providing arguments for having several constraints on the system regarding the context of its development. These constraints mainly arose from the fact that the software product has to be developed over a (short) period of ten weeks, and within the resources IBM provides, firstly this is elaborated upon in section 2.5.1. Second, in section 2.5.2, each of the hardware choices is explained. Third, in section 2.5.3, each of the software choices is explained.

2.5.1. HARDWARE CONSTRAINTS

Choices concerning the hardware had to be made well in advance for two reasons. First, the time span of the project is relatively short in comparison to the delivery time for affordable hardware. Secondly, it takes time to test and get used to conventions regarding the hardware and the software it is programmed. Ordering of hardware in later states of the project would incur increased time pressure, and thus frustration within the development and unnecessary dependencies on external parties for the success of the project.

The context constraints can not be seen as design goals (listed in section 2.3), as they do not form a goal that has to be accomplished, neither can they be seen as requirements (listed in section 2.4), as they are not aimed at realizing a certain design goal. The context constraints namely form a collection of agreements the development team members made with the technical and research supervisors from IBM, respectively Manfred Overmeen and Robert-Jan Sips.

2.5.2. HARDWARE CHOICES

RASPBERRY PI'S

The Raspberry Pi[6], a low-cost, complete computer, will be used for routing, and, if necessary, sensing. This choice was motivated by their low cost, energy efficiency, the enormous amount of (educational) documentation, and the ability to easily extend them via modules with extra functionality (such as WiFi or Bluetooth communication capabilities). Extended with a storage device, they can function as buffer between (low-cost) sensing devices and the cloud application.

ARDUINO'S

The Arduino, a low-cost micro-controller, will be used as primary sensing device in the network. This choice was motivated by their small size, low cost, energy efficiency, the enormous amount of documentation, and the ability to extend them with multiple sensor easily through simple electronics. Furthermore, with the addition of a WiFi or Bluetooth module, they can be used wireless. These factors make it an ideal candidate to be spread over an office to sense phenomena such as temperature.

BLUEMIX

At its roots, IBM is market leader in business-grade hardware, especially servers. They offer practically unlimited access to computer power and storage, which makes the purchase or renting of servers not required. Furthermore, they offer a clean interface, BlueMix, which directly offers a variety of services including database and web server services. Bluemix is also a portal to IBM Watson, which allows could support semantic interpretation of produced data.

[6]As can be found in chapter 4.3 The Raspberry Pi is not used for implementing a hub node in the eventual system

2.5.3. SOFTWARE CHOICES

SMARTPHONE APP

The smartphone application will first be created in Android. This is motivated by previous experience with programming applications for Android by the development team, and the high presence of Android smartphones within the IBM office.

CLOUD APPLICATION

The cloud application requires two services: a database and a web server. Both are offered by IBM Bluemix. A NodeJS web server is chosen due to its simplicity in creating endpoints, and the data model (JSON) it handles is easy to encode. A relational database will most likely be used for storage of data, because of the experience of the development team through courses to use and optimize it.

ARDUINO

The official documentation of Arduino uses the Processing language, which is supported by the Arduino environment they provide. Furthermore, Processing is natively supported by the boards, and is capable to access the complete functionality offered by the Arduino board.

RASPBERRY PI

The Raspberry Pi is commonly used with the Linux operating system, due to its small system footprint. The Linux OS Raspbian, part of the NOOBS software package, supports Python, the official full programming language for Raspberry Pi's. Most documentation and example projects are written in Python. For these reasons, Python will be used[7].

2.6. DEVELOPMENT METHODOLOGY

The key components of the methodology adopted in the process of developing the system are described in this section. Most of the adopted techniques came forth from our academical experience with software projects. The version of Scrum that was used throughout the process is described in section 2.6.1.

2.6.1. ADAPTED SCRUM

The fundamental principles of Scrum enable efficient, and effective software development, as we can state based upon our experience of this software development method. The success of Scrum comes from the fact that it doesn't blame the customer for giving the wrong requirements, but it accepts that "project requirements might be unclear or unknown even as the project gets underway" [22]. The same holds for our project. We do know its main goals and the general deliverables, but the requirements might change throughout the process due to new findings done along the way. We therefore chose to adopt this software development methodology.

Due to the small size of the development team, the regular Scrum roles were not explicitly appointed. The continuous collaboration in close proximity, and the small team size, allows us to effectively maintain the product backlog collaboratively, and fuel the capability to accomplish the product goals and provide the deliverables. Taking into account the two month duration of the project, and the (ambitious) goals that are set, the sprint distance is set to one week. This results in having at least one meeting with our stakeholders, the TU coach and client, every week, in order to appraise them of recent progress and discuss the next steps in the process. A sprint review is held at the end of every sprint to reflect on progress, and to determine the sprint backlog for the next sprint. At the start of each day the tasks to be done will be discussed within the development team, and a rough planning of the day is decided. At the level of sprints, evolutionary prototyping [23] will be applied, as this would enable the agile development of the product.

[7] Raspberry PI's were investigated, but not used in the demonstrator. A closer study of the NOOBS operating system unveiled a default support for Java. This means that the existing hub node code could work on the Raspberry PI, provided that a supported Bluetooth stack is present on the device

3

DESIGN

3.1. INTRODUCTION

This chapter is devoted to the design of the overall system and its components, and the backing of the main design choices that were made. A high level overview of the system design, and the responsibilities of the four components are given in section 3.2. Furthermore, in section 3.3, the data model of the system is described via the Entity Relationship modeling method. For comprehensibility, a distinction is made between the main design corpus described in this chapter, and the eventual implementation, which is described in chapter 4.

3.2. ARCHITECTURE

In this section the architecture of the overall system is discussed. First, in section 3.2.1 a top-level overview of the system is given, introducing its four core components. Second, the responsibilities of the sensor node (section 3.2.2), the hub node (section 3.2.3), the server (section 3.2.4), and the database (section 3.2.5) are described. In section 3.2.6 we provide the reader with argumentation for the introduced architecture by backing the design choices that were made.

3.2.1. OVERVIEW

A scalable distribution of system intelligence was required in order for the system to function, regarding e.g. maintainability and safety. The design follows the design principle of maintaining low coupling and high cohesion. This means that there is low interdependence between components, while highly related functionality is put together as closely as possible (within a component). Two main responsibilities were identified in the requirements, namely (1) the gathering of (sensor) data from low-cost and smartphone sensor nodes on-site, and (2) the persistent storage of data in a central database. To effectively distribute system intelligence to support these responsibilities, the choice was made to divide the system up into four components: the sensor nodes, hubs, server and database. An argumentation for this architecture design can be found in the first two subsections of section 3.2.6. The system overview is depicted in figure 3.1.

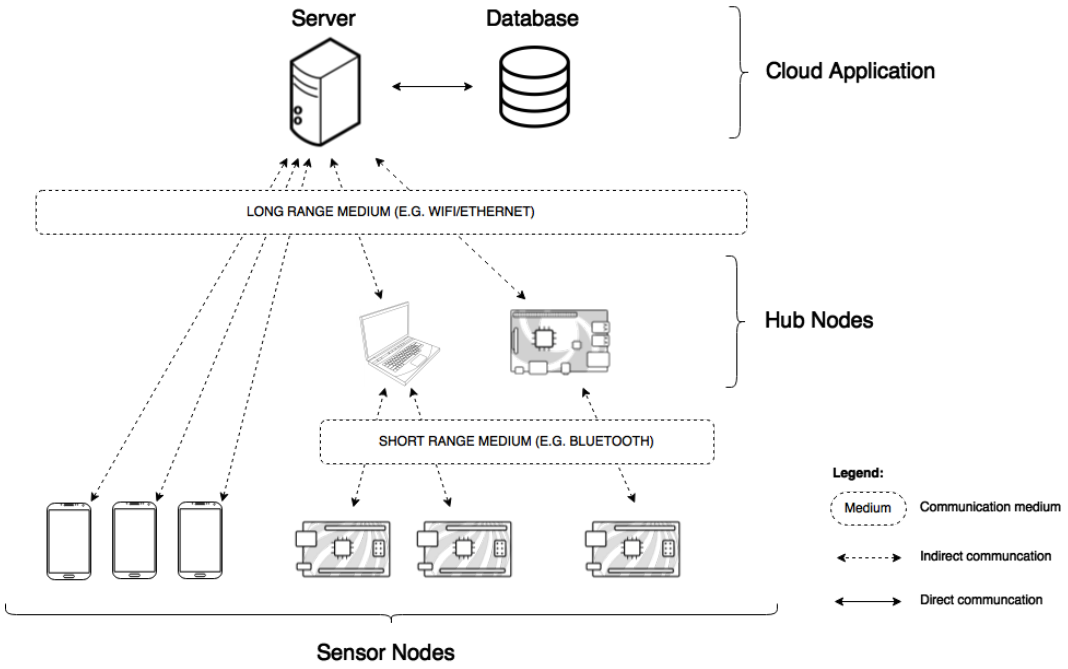

Figure 3.1: System Overview

In figure 3.1, a hierarchy from top to bottom is shown. The hierarchy consists of three layers (sensor nodes, hub nodes and cloud application) containing four components (sensor and hub nodes, the server, and the database) in total. These components are represented by six types of 'devices', as depicted by the different images.

The sensor nodes are depicted at the bottom layer, and consist out of two types of sensing devices. On one hand there are sensor nodes with long range communication capabilities (represented by the smartphones on the left side). These sensor nodes communicate directly with the server through a long range medium such as by accessing the Internet through WiFi. On the other hand there are the sensor nodes with only short range communication capabilities (represented by the Arduino boards on the right side). These sensor nodes do not communicate directly with the server, but rather use their short range communication capabilities (e.g. Bluetooth) to communicate with hub nodes.

The hub nodes (depicted at the middle layer) in turn send the data they received from the sensor nodes to the server through a long range communication medium such as accessing the Internet through Ethernet or WiFi. This can be done by any device with matching both short range and long range communication capabilities (i.e. a laptop or Raspberry Pi). The hub nodes act as channels between the server and sensor nodes which solely have short range communication capabilities. The hub manipulates the data it passes on as little as possible, only accounting for the inconsistencies its presence as a channel induces (e.g. delay).

The server (depicted left at the top layer) configures the sensor nodes and hubs, and receives sensing data from all of the sensor nodes. The server stores this received data in the database it is linked to. This database (depicted right in the top layer) forms a central storage point for all configuration and sensing data.

3.2.2. SENSOR NODE RESPONSIBILITIES

The sensor nodes are the on-site devices that have a sensing capability, unlike the other components of the system. Its responsibilities are related to the gathering of reliable measurements, and transmitting them to the server. There is a difference between sensor nodes that communicate directly with the server, and ones that communicate indirectly via a hub (respectively depicted left and right in figure 3.1). Nevertheless, their responsibilities are equal:

Measuring The sensor node continuously produces measurements through its sensors. It stores these measurements with their time, value, and sensor origin.

Buffering The sensor node continuously gathers measurements, which need to be stored before transmission. If there is (temporarily) no network connection, it must buffer those measurements in a first-in-first-out fashion when the buffer's size limit is reached.

Sensor maintenance The sensor node checks whether the sensors are still working properly. If the sensor detects defects, it should remove that sensor from its configuration.

Persistent storage The sensor node must persistently store the identification and configuration data it received from the server. This ensures that when the device loses power, it is not detected as a new device by the server.

Transmission The sensor node itself is responsible for communicating with the server, this includes identification, (dynamic) configuration, and passing its measurements. This transmission can either be done directly to the server (e.g. via WiFi) or indirectly via a hub (e.g. via Bluetooth).

3.2.3. HUB NODE RESPONSIBILITIES

The hub nodes are the devices on-site that have both short and long range communicative capabilities, allowing them to be a channel between the server and the sensor nodes, of which the latter solely has short range capabilities. The responsibility of the hub node focuses on reliably passing on messages between the server and sensor nodes. Its responsibilities are the following:

Message channel The hub node must transmit any messages it receives from sensor nodes to the server, and vice versa. Processing of these messages is done at the server, therefore the hub is mostly a bridge in the communication pathway between the low-cost sensor nodes and the server.

Traceability The hub node must identify itself with the server. It should then always attach its identification information to the meta-data of the messages it channels, offering the server the ability to monitor the performance of hubs.

Temporal consistency The hub node should channel messages "*as if it was not there*". To achieve this, it must calculate the delay it induces between receiving the message and passing it on. This delay must then be included in the meta-data of the message.

3.2.4. SERVER RESPONSIBILITIES

The server is seen as a single point with which the sensor nodes can communicate (through a hub if necessary). The server is generally responsible for providing services to sensor nodes, and allowing the monitoring of the system by its administrators. In this service it maintains semantic data consistency, and supports configurability of the system. The server has the following responsibilities:

Identification The server allows (both sensor and hub) devices to request a system-wide identification. This received identification can then be used to uniquely link any future correspondence with the server to a single device in the system. The identification request includes device type and initial location.

Configuration registration The server allows sensor nodes to register their own device configuration with the server. This configuration contains the sensors that are on-board of the sensor node. This service can be called upon multiple times by a device (e.g. due to sensor failure, or change).

Measurement The server allows sensor nodes to persistently store measurements in the system it measured in both a temporal and spatial dimension. The service is responsible for handling delays.

Dynamic configuration The server allows system administrators to dynamically configure the sensors on the sensor nodes. For example adjusting the interval at which the sensors produce measurements, or whether they are active.

State preservation The server stores the state of the system at any point. This means that any change in (dynamic) configuration is stored in the database as well, not just the effect of it on the momentary system state.

3.2.5. DATABASE RESPONSIBILITIES

The database is an organized collection of relational data, containing both measurement-oriented data the system produces, as well as state-oriented data (e.g. device types, devices, sensors, sensor activity). Its responsibilities are related to the storage of data, the presentation of data, and maintaining the relational data constraints. The responsibilities are the following:

Data Storage The database continuously receives insertion requests from the server. The database needs to efficiently store this data, while ensuring that imposed relational constraints are upheld (as an insurance for data consistency preservation).

Data Retrieval Many data constraints are left to the server due to limitations of relational constraints (see relations in section 4.5) and practical considerations (e.g. it is better to check for the existence for an identifier, rather than blindly inserting and performing error handling). The server will frequently query the database to check these constraints, thus it is the responsibility of the database to answer promptly (e.g. by maintaining additional indexes).

3.2.6. BACKING OF DESIGN CHOICES

In this section we provide the reader with argumentation for the design choices that were made in translating the requirements and context constraints (described in section 2.4 and 2.5 respectively) into an actual design (described in this chapter). First, in section 3.2.6, an argumentation is given for the addition of a hub node. Second, in section 3.2.6, the need for a central server is elaborated upon. Finally, in section 3.2.6, the chosen push/pull mechanisms are described.

HUB ADDITION

As mentioned in section 3.2.1, the sensor nodes can communicate directly with the server or indirectly, through a hub. There are four factors that accounted for the addition of this hub:

Device Exclusion Devices that cannot be extended with long range communication capabilities would be excluded from participating in the system.

Network Restrictions In certain environments, addition of unknown devices to the WiFi network is not possible. In the case of IBM, a device firstly has to be authenticated and approved by the network administrator. Safety concerns for the internal infrastructure made this approval process troublesome.

Extension Cost Additional costs will be made for devices that can be extended with an extra long-range communication module. Furthermore, embedded system devices typically do not possess an UI or a browser, nor the protocol stack to handle complex authentication procedures (which was the case with the cheap ±$5 ESP8266 for the Arduino). The added module would need to support all these features on top of the embedded system device, which will increase the price significantly (e.g. an Arduino WiFi shield of ±$60).

Existing Infrastructure The mass presence in the work environment of laptops with both short-range (Bluetooth) and long-range (WiFi) communication capabilities are already ideal candidates to become hubs (as is the case of IBM). These laptops already have network access, thus circumvent any possibly imposed network restrictions.

These factors proved the hub to be an elegant solution for the system. It would namely be the best solution with respect to the design goals we value: installability, extendability, and deployability.

CENTRAL SERVER

To meet the demand of central storage, a central component where all data is stored - a database - is required. A choice was made to put a server in front of this database, allowing other components to communicate with the server instead of directly communicating with the database. This choice was motivated by four arguments. First, the other components only need to talk the language of the server, which we can control, instead of relying on the choice of database and its interface. Second, in order to guarantee consistency and reliability of data stored in the database, an application-specific implementation is required to check the semantic constraints. Third, to allow the system to be adaptable and configurable, a server can instruct devices that connect to it, whereas a database would only work as a passive servant to the other components. Fourth, it abstracts the sensor nodes from the choice of database, and enables the system to be scaled up by only adapting the server.

PUSH/PULL MECHANISMS

The management of the energy consumed by system components was taken into account for creating the system design. The sensor nodes are commonly wireless, thus do not have a continuous power source. The server does possess this ability, and the hub (often) does so too, or at least has a larger power supply. This supported the design choice to let the devices themselves determine when to communicate with the server. The hub in this equation actively offers sensor nodes the ability to do so and engages in the connection, whereas indirect sensor nodes merely accept or reject. This reduces cost of energy consumption at the side of sensor nodes.

3.3. DATA MODEL

This section is devoted to explaining the design choices made when creating the data model for the cloud application. First, in section 3.3.1, the methodology is laid out. Second, the requirements from a measurement centric point of view are described in section 3.3.2. Third, the requirements from a device centric point of view are described in section 3.3.3. Finally, these requirements are expressed in an Entity Relationship Model in section 3.3.4, including a visual depiction in the form of a diagram.

3.3.1. DATA MODELING METHODOLOGY

The research phase (chapter 2) was the first step in the creation of a data model that could support a system specified by the requirements (see section 2.4). These requirements, however, were too abstractly defined to function as direct data model requirements. To narrow this 'abstraction gap', further interviews and discussions were held with the two stakeholders that were the closest related, Robert-Jan Sips and Uditha Ravindra. The former has had significant experience with data modeling, and was able to most clearly communicate his needs. The results of these stakeholder interactions allowed for a more precise composition of data model requirements.

During the development, the data model has undergone several iterations. After each iteration, it became more fit for the system's needs, both in terms of technical and stakeholder demands. The reasoning behind the requirements is added as a story after each requirement category. In these stories, a focus was put upon the most important choices. A separation was made between measurement and device centric requirements (respectively laid out in section 3.3.2 and 3.3.3). These data model requirements were then translated using entity relationship (ER) modeling in section 3.3.4 into an ER model, including a visual diagram.

3.3.2. MEASUREMENT CENTRIC REQUIREMENTS

Measurement Centric List

- **1a** The (synchronized) time of a measurement must be known;
- **1b** The location of a measurement must be known;
- **1c** The location of a measurement must be traceable to the location of a sensing device;
- **1d** A measurement must be traceable to a specific device configuration at a specific point in time;
- **1e** New measurement types (from new sensor types) must be able to be dynamically assigned and subsequently measured;
- **1f** A sensor of the same model can measure multiple quantities;
- **1g** A sensor can produce multiple measurements;
- **1h** A measurement is performed by one sensor in one specific device configuration;
- **1i** Sensors are either active or not;
- **1j** Absence of measurements due to sensors being deactivated must be traceable
- **1k** A measurement must only be in one environment;
- **1l** Multiple environments can exist in which measurements are taken;
- **1m** Environments must have a unique name;
- **1n** Environments can be in different time zones;
- **1o** The locations of measurements are relative to the origin of the environment;

1p The hubs that pass on measurements must be traceable;

1q The type of device that the hub is must be stored;

1r Sensing devices need to able to switch between hubs freely;

1s New device types need to be able to be added dynamically;

Measurement Centric Story
In order to be able to distinguish between different measurements, the time and location of a measurement must be known (1a,1b) and the location of such a measurement must be traceable to the location of a sensing device (1c) and a specific device configuration of a sensing device (1d). But, you can imagine that new types of sensors are added to sensor nodes (think of a newly added sensor to an Arduino or smartphone). Therefore we want the potentially new measurement types of these sensors to be dynamically assigned and subsequently measured, so that the system enables sensor extension (1e). Ofcourse a sensor model can measure multiple quantities and a sensor can produce multiple measurements, in which each measurement is performed by one sensor in one specific device configuration (1f,1g,1h). But, we do not always want the sensor nodes to measure everything they can (according to their device configuration). We would rather want to be able to tell them when to measure what (1i) and archive this (1j). A measurement always takes place in a certain environment (like the second floor of IBM CAS in Amsterdam, or the second floor at the faculty of EEMCS at TU Delft) (1k,1l,1n), which has a unique name (1m) and is located in a certain time zone (1m). The locations of measurements are stored relative to the origin of a certain environment (1o). The data from measurements is sent to the cloud through hubs. We want this transfer of data to be traceable (1p,1q) and dynamic (1r), and to allow for the adoption of new device types (1s).

3.3.3. DEVICE CENTRIC REQUIREMENTS

Device Centric List

2a A sensing device is uniquely identified by a device identifier;

2b A sensing device must have a name;

2c The location of the sensing device, including its sensors, is registered as a singular point in three dimensions;

2d The timestamps the sensing device provides in its communication, must be synchronized with the general time of the system;

2e A sensing device always has a device configuration;

2f A sensing device can not have multiple device configurations at the same time;

2g The history of previous sensing device configurations of a device must be stored;

2h A sensing device configuration consists out of zero or more sensors;

2i A (physical) sensor can only be added to a single device configuration;

2j A sensing device configuration can exist out of multiple sensors of the same type;

2k Multiple sensors can have the same type;

2l A sensor can have only one sensor type;

2m A sensor type can exist while there are no measurements of that type (e.g. when a device with a 'unique' sensor announces its configuration but immediately drops out afterwards);

2n A sensor type has a model, quantity, unit, default interval, and default active setting of new sensors;

2o A device type has a brand (e.g. 'Samsung'), model (e.g. 'SM-G920'), marketing name (e.g. 'Galaxy S6') and an category (e.g. 'smartphone');

2p A device type is uniquely identifiable using its brand and model.

Device Centric Story
In order to distinguish between devices, every sensing device has a unique device identifier (2a) and for the ease of usage, we also gave each device a name (e.g. 'CAS Bureau #1') (2b). In order to couple measurements to both time and location, these variables must be registered for the sensing device (2c,2d). In order to know what measurements can be done by which device, we store its device configuration (2e,2f). The client

pointed out that he wants the history of previous sensing device configurations to be archived (2g). A device configuration can consist out of multiple sensors (2h), but a (physical) sensor can only be added once to a single device configuration (2i), whereas a device configuration can consist out of multiple sensors (2j) that can have the same type (2k). Every sensor can have only one sensor type (2l), but this does not mean that the database will contain measurements of every type (2m). A sensor type is chosen to consist of a model, quantity and unit of measurement (2n), and a device type is chosen to consist of a brand, model, marketing name and category (2o). A device type is thereby uniquely identifiable using its brand and model (2p).

3.3.4. ENTITY RELATIONSHIP MODEL

From the requirements in the previous sections, ten entities were identified[1]: SensingDevice (s), SensingDeviceConfiguration (w), Sensor (w), SensorActivity (w), SensorType (s), Measurement (w), SensingDeviceLocation (w), Environment (s), Hub (s), and DeviceType (s). Twelve relationships bind these strong and weak entities together. Not all requirements can be (wholly) expressed through the ER model, due to its expression limitations. This ER model enables quick transfer of the concept of the data model, as it covers the necessary main components fundamental to understanding the system. Requirements that are expressed, are linked in the sentences by adding their number between brackets, e.g. "(1a)". A visual representation of the ER model, the ER diagram, is shown in figure 3.2.

SensingDevice A strong entity, it represents a sensing device in the system. It is identified using a device identifier (2a). It has three other attributes: LastConfigTime, Name (2g), and LastLocation (2l). It must be coupled to at most one device configuration in time (2b). The ActivatedTime attribute of the ConfiguredWith relation enables placement of a configuration in the time domain (2e).

SensingDeviceConfiguration A weak entity, it represents the (history of) sensing device configurations (2d). It is uniquely identifiable through the combination of ActivatedTime and DeviceID (2c), both obtained from the ConfiguredWith relationship. This combination is however not used as primary key, to avoid creating string identifiers which take a significant larger space than a numerical identifier when often referenced. It has zero or more Sensor entities related through the ConsistsOutOf relationship.

Sensor A weak entity, it represents a sensor attached to a device in a specific configuration. It has one attribute: Name (e.g. "NORTH"). It is uniquely identifiable through the combination of its Name attribute (2j) and the two related entities, through IsSensorTypeOf and ConsistsOutOf relationships, namely SensorTypeID and ConfigID. It must be related to one SensorType (2l), and one SensingDeviceConfiguration (2h, 2i). It must have an entry in the SensorActivity table (1m). It can create multiple measurements through the MeasuredBy relationship (1d).

SensorActivity A weak entity, it represents whether a sensor is active or not in time. It is identified through the ActivityLog relationship with Sensor (1m). It has two attributes, namely Active and Time.

SensorType A strong entity, it represents the types of sensors that exist in the system. It has six attributes: SensorTypeID (generated identifier), Model (e.g. "KY001"), Quantity (e.g. "TEMPERATURE"), Unit (e.g. "CELCIUS"), DefaultInterval (e.g. 5000), and DefaultActive (e.g. true) (2n). The combination of Model and Quantity is unique, but SensorTypeID is used as identifier to avoid creating string identifiers which take a significant larger space than a numerical identifier when often referenced. It defines the type of zero (2m) or more sensors through the IsSensorTypeOf relationship (1b, 2k).

Measurement A weak entity, it represents a measurement in the system. It is uniquely identified through the MeasuredBy relationship with Sensor (1c, 1e), using the combination of SensorID, and its own attributes Tag and Time. It has three attributes, namely Value, Tag and Time (1a). Its location and environment are determined through the MeasureAt relationship (1o, 1p, 1r, 2e). The hub that passed it on (if there is one) is determined by the AggregatedBy relationship (1j, 1l).

SensingDeviceLocation A weak entity, it represents the location of a SensingDevice at some point in time. It is uniquely identified through the LocationID. There is however a unique combination of possible through the LocatedAt relationship, using the combination of Time and DeviceID of that relationship. It has two attributes: LocationID (generated identifier) and Location (e.g. (x: 4.2, y: -1.2, z: 1)) (2e). It determines its environment through the MeasuredIn relationship.

[1] (s): strong entity; (w): weak entity

Environment A strong entity, it represents a measuring environment. This is the only entity that is created by the administrator of the server. It has four attributes: EnvID (generated identifier), Timezone (e.g. “GMT+1”) (1g), Name (e.g. “IBM CAS”) (1h), and origin (e.g. latitude, longitude, elevation) (1i), which is the location to which all measurements in the environment are taken relative to. Zero or more measurements are linked to the environment through the MeasuredIn relationship (1f).

Hub A strong entity, it represents a hub which passes on communication from sensing devices to the server. It has two attributes: HubID (generated identifier), and Name (e.g. “PI IBM CAS #1”). The Name is purely for identification by humans, to the server a hub is just used with its identifier. The type of device that the hub is, is determined by the IsHubDeviceTypeOf relationship (1k).

DeviceType A strong entity, it represents a type of device (1q). It has four attributes: DeviceTypeID (generated identifier), MarketingName (e.g. “Samsung Galaxy S3”, “Arduino Uno”, or “Raspberry PI 2 B+”), Brand (e.g. “Samsung”, “Arduino”, or “Raspberry PI”), Model (e.g. “SHV-E210L”, “Uno ATmega328P”, or “B+”), and Category (e.g. ’smartphone’, or ’board’) (2o). The combination of Brand with Model is unique, but DeviceTypeID is used as a dedicated identifier to avoid creating string identifiers, which require a significant larger amount of memory than a numerical identifier.

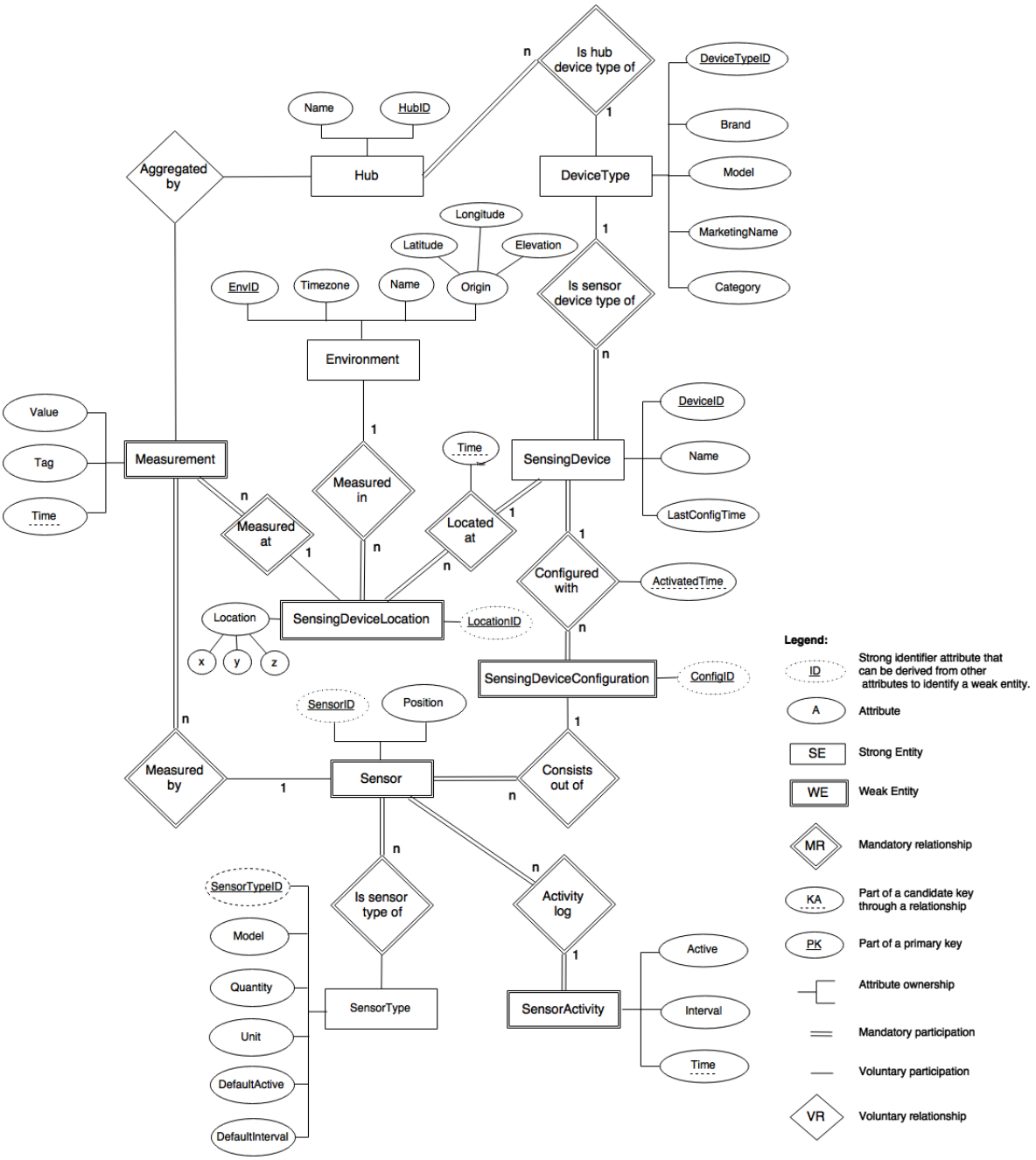

Figure 3.2: ER Diagram

4

IMPLEMENTATION

4.1. INTRODUCTION

This chapter is devoted to the implementation of the system. The implementation is based on the design described in chapter 3, in which it was specified that the system consists of four components. For each of these components their implementation is discussed in a separate section in this chapter. The first three sections describe the hardware used in the implementation, the dependencies regarding the software (e.g. libraries, tools and their licensing), the design of the software (e.g. pseudo-code) and the actual implementation (e.g. class diagrams) regarding each component of the system. First, in section 4.2, the implementation of the sensor node is discussed for both the low-cost sensor node and the smartphone app. Second, in section 4.3, the implementation of the hub node is elaborated upon. Third, in section 4.4, the implementation of the server is described. Finally, in section 4.5, the implementation of the data model in a relational database is described.

4.2. SENSOR NODE

The sensor node component has been implemented on two platforms for the demonstrator. In this section, those implementations are discussed. First, in section 4.2.1, the chosen hardware of the two platforms is discussed. Second, in section 4.2.2, the platform-specific software dependencies are explained. Third, in section 4.2.3, the platform-independent software design is elaborated upon.

4.2.1. HARDWARE

The sensor node component in the system architecture has been implemented on two platforms: Arduino and Android. First, in section 4.2.1, the choice for the Arduino hardware is explained. Second, in section 4.2.1, the choice for the Android platform is elaborated upon.

ARDUINO HARDWARE

The Arduino board is immensely popular among electronics enthusiasts, having produced over 700.000 official boards and roughly the same amount in clones[1]. With the goal being a learning platform, its documentation[2] is in excellent shape, enabling developers to quickly comprehend the features it offers. These include all the features required for a sensor node, including timing, dynamic addition and removal of sensors, both persistent and RAM memory, and, with addition of a module, communicative capability. Three boards have been tested in the demonstrator: the Arduino Nano, Uno, and Mega 2560. Because of its popularity, libraries have already been written for many of the additive electronics (e.g. sensors) that Arduino supports. The additive hardware chosen to complement the Arduino boards in becoming a sensor node, is listed in table 4.1.

The choice of sensors (KY001, DHT11, KY018) for the demonstrator Arduino board was based on both the availability of libraries, the phenomena they measure (matching the requirements), and the range and accuracy they posses. For the system design itself, the sensor is merely seen as a component providing a value, making the choice for sensor of significantly less importance in a demonstrator of the system.

[1] http://medea.mah.se/2013/04/arduino-faq/ (retrieved 6 June 2015)
[2] http://www.arduino.cc/en/Reference/HomePage (retrieved 6 June 2015)

There is a wide variety of communication modules, namely so-called *shields* that are the same size as an Arduino, and smaller stand-alone modules, which connect via a serial port. The former were outside of the price range, being even more expensive than the board itself and all of its sensors combined. This supported the decision to disregard this option. Two stand-alone modules were examined, namely the ESP8266 WiFi module, and the HC-05/06 Bluetooth module. The former, the ESP8266, proved to be unreliable in data transmission, often resulting in data loss. The latter, the HC-05/06, proved to reliably send and receive data in our conducted tests. Therefore the HC-05/06 was chosen.

All three boards (Nano, Uno, and Mega) are able to handle the additive hardware selected. An electronic schematic for the smallest, the Nano, is depicted in figure 4.1.

Model	Cost	Functionality	Specifications	Library exists
HC-05	$3.90[3]	Bluetooth module (digital / serial port)	2.4GHz, 3Mbps[4]	None required, used via Serial Port Profile (SPP)
KY001	$2.12[5]	Temperature sensor (digital)	-55°C to +125°C, range accuracy +- 0.5°C between -10°C and +85°C [6]	Yes, MAX31850 DallasTemp[7] and OneWire [8]
DHT11	$0.99[9]	Humidity sensor (digital)	20% to 80% humidity, range accuracy 5%[10]	Yes, AdaFruit DHT[11]
KY018	$1.66[12]	Light intensity sensor (analog)	0 (light) - 1024 (dark)	None required, provides analog value

Table 4.1: Arduino Additional Hardware

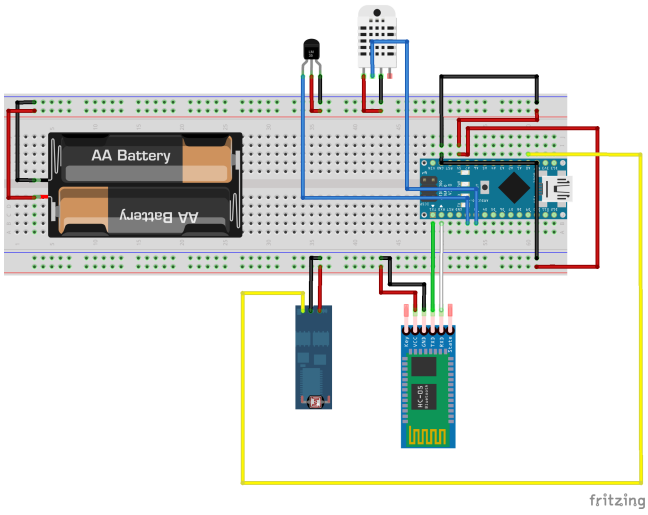

Figure 4.1: Electronic schematic for the Arduino Nano sensor node (generated using Fritzing)

[3] http://www.ebay.com/itm/Wireless-Serial-6-Pin-Bluetooth-RF-Transceiver-Module-HC-05-RS232-Master-Slave-/400562862516 (retrieved 6 June 2015)
[4] http://wiki.iteadstudio.com/Serial_Port_Bluetooth_Module_%28Master/Slave%29_:_HC-05 (retrieved 6 June 2015)
[5] http://www.ebay.com/itm/KY-001-DS18B20-Temperature-Sensor-Module-Temperature-Measurement-Module-Arduino-/271643288650 (retrieved 6 June 2015)
[6] https://tkkrlab.nl/wiki/Arduino_KY-001_Temperature_sensor_module (retrieved 6 June 2015)
[7] https://github.com/adafruit/MAX31850_DallasTemp (retrieved 6 June 2015
[8] https://github.com/adafruit/MAX31850_OneWire (retrieved 6 June 2015)
[9] http://www.ebay.com/itm/1pcs-DHT11-Digital-Humidity-Temperature-Sensor-NEW-/171302145362 (retrieved 6 June 2015)
[10] https://www.adafruit.com/products/386 (retrieved 6 June 2015)
[11] https://github.com/adafruit/DHT-sensor-library (retrieved 6 June 2015)
[12] http://www.ebay.com/itm/KY-018-photoresistor-module-for-Arduino-AVR-PIC-/121167420768 (retrieved 6 June 2015)

ANDROID HARDWARE

The Android operating system, unveiled by Google in 2007, has a smartphone market share of 78% as of Q1 of 2015[13]. Android's source code is licensed under open source licenses, while also allowing it to be combined with proprietary software which is the case for this project. Its documentation is vast, evolving itself for the better with each iteration (the latest being Android 6.0: M). One of its keystones is backward compatibility, offering libraries to both support older platforms, and still use the latest technologies it has to offer. This maximizes the range of devices that an Android application can reach. Because of major changes between 2.1 and 2.2, any device that runs Android 2.2 or higher will be supported by the application. Smartphones such as the Nexus 5 or Samsung Galaxy S5 house a wide variety of sensors, e.g. battery temperature sensor, light sensors, and microphones.

Other options were also explored, such as IBM MobileFirst, and native iOS. During the exploration phase we found that the former, IBM MobileFirst, was not as thoroughly developed and adopted by programmers, compared to other platforms. Due to a lack of equipment to develop for iOS, it was found to be impossible in the scope of this demonstrator to develop an iOS application. However, the logical design in section 4.2.3 outlines how to develop for other platforms, such as iOS, in future iterations. Moreover, to support cross-platform deployment, the Android-specific code (primarily concerning interface) and the general Java code have already been kept as separated as possible following the model-view-controller design pattern.

4.2.2. SOFTWARE DEPENDENCIES

To program the hardware chosen in section 4.2.1, choices in software were made for both platforms: Arduino and Android. First, in section 4.2.2, the choice of software for the Arduino hardware is explained. Second, in section 4.2.2, the choice of software for the Android hardware is explained.

ARDUINO: C++/PROCESSING

C++ and Processing are the official languages of the Arduino platform. They combine the strength of Arduino specific functionality, and C++'s vast library functionality. This includes low level memory access, abstraction through classes, and native code running at high speeds. Complimentary libraries are added for the sensors, and testing. The ArduinoUnit framework is chosen due to its little boilerplate code, and its small size. All selected libraries are depicted in table 4.2 in combination with their licensing.

Name	**Functionality**	**License**
Arduino	Arduino specific functionality such as timing, and reading/writing to ports.	GNU LGPL
ArduinoUnit	Unit testing framework.	MIT License
OneWire	Support for slave devices that operate using a 1-wire protocol.	MIT License
DallasTemperature	Support for temperature measuring devices that use the 1-wire protocol.	GNU LGPL v2.1 or higher
DHT Sensor Library	Supports the reading of the DHT*xx* series sensors.	MIT License

Table 4.2: Arduino libraries

ANDROID: JAVA

Java is the native language of the Android platform. It abstracts from device specific code by using a virtual machine (DalvikVM). Through this abstraction, it has proven to be virtually device-independent, while still offering low-level access to the hardware in the device, e.g. the battery temperature sensor. Usage of Java also enables all the functionality that a platform offers, including multithreading, encryption, and interfaces. Compatibility is central in its complete official documentation, with many online communities such as StackOverflow offering answers to implementation questions. For unit testing the well-known and well-documented JUnit framework is used, with the latest version that is compatible with Android (v3). EMMA is chosen as coverage tool, as it is already present on the Android platform, and continues to receive support from Google. All selected libraries are listed in table 4.3.

[13] http://www.idc.com/prodserv/smartphone-os-market-share.jsp (retrieved 6 June 2015)

Name	Functionality	License
Android SDK	Android specific functionality	SDK License
JUnit 3	Unit testing framework	Eclipse Public License v1.0
EMMA	Coverage Tool	Common Public License v1.0

Table 4.3: Android libraries

4.2.3. SOFTWARE DESIGN

Independent of any platform, there is a general logic design for the sensor node. This has been implemented on both chosen platforms using its platform-specific software described previously (in 4.2.2). First, the high level pseudo-code is given as an overview to its program flow. Second, the different operational modules on which the program flow relies, are discussed. Third, each phase in the program flow is elaborated upon.

HIGH LEVEL PSEUDO-CODE

Listing 4.1: Sensor Node High Level Pseudo-code

```
input: communication, storage, sensors, measurements

begin
    while (true) {

        if not identified then
                identify()
        else if not configured then
                configure()
        else if not dynamically configured then
                archive() // Zero measurements is flag for dynamic configuration
        else
                sense()
                if measurements.size > THRESHOLD then
                        archive()

    }
end
```

OPERATIONAL MODULES

Communication The module responsible for communication with the server. This is typically implemented blocking in single-thread environments. It receives an endpoint and message from the caller, passes this message via a POST request to the respective server endpoint (direct or indirect via a hub), and then returns the server's response to the caller.

Storage The module responsible for persistent storage of variables on the sensor node. It can store and retrieve the identification and configuration.

Sensors Fixed size array of sensors that are on-board of the sensor node. Each element is an abstraction module for a physical sensor. It keeps track of the value of the physical sensor it wraps, and saves the interval and activity of the sensor's measurement production.

Measurements Buffer for the storage of measurements. Each measurement contains a sensor node timestamp, so that the original time of measurement can be calculated.

PHASES

identify() Checks whether an identification (i.e. device identifier) exists in the persistent storage. If it does not, it requests a new identification via the communication from the server (giving its device type and initial location) and stores it persistently.

configure() Checks whether credentials for a configuration exist in the persistent storage. If one exists, it checks if it is still up-to-date with the device's configuration at boot-up. If it is either not stored, or not up-to-date, it will submit its new configuration (giving its sensor types) to the server, and will afterwards receive credentials for it (i.e. sensor identifiers).

archive() Composes a request of all the measurements stored in the buffer, along with their time (delay) and location. Sends this request to the server. It will however also handle any dynamic configuration messages it receives such as intervals and activity of sensors.

sense() Goes over all sensors, and asks them where they want and can submit a measurement. If a sensor wants to and can do so, it retrieves the value, resets the timer of the sensor, and adds a new measurement containing this information to the measurements buffer.

4.2.4. SOFTWARE IMPLEMENTATION

In this section we elaborate upon the software implementation of the sensor node. For the Android code we discuss their behavior and interfaces separately, as described in the first two following sections. At the end of this section, the Arduino software implementation is discussed.

INTERFACES VERSUS BEHAVIOR

To separate platform specific code (interface) from sensor node code (behavior), a clear separation is made in the implementation of the Android platform. The interface code mostly calls Android functionality, for tasks related to user interaction (e.g. starting/stopping the sensor node, or input of device name). The behavior code is more general and concerns the platform-independent software design laid out in section 4.2.3. The latter could easily be adapted to support other Java-based operating systems.

ANDROID BEHAVIOR IMPLEMENTATION

The behavior is visually depicted using a UML diagram in figure 4.2. The *DeviceBehavior* class is the connector with the interface implementation, and runs on a separate thread.

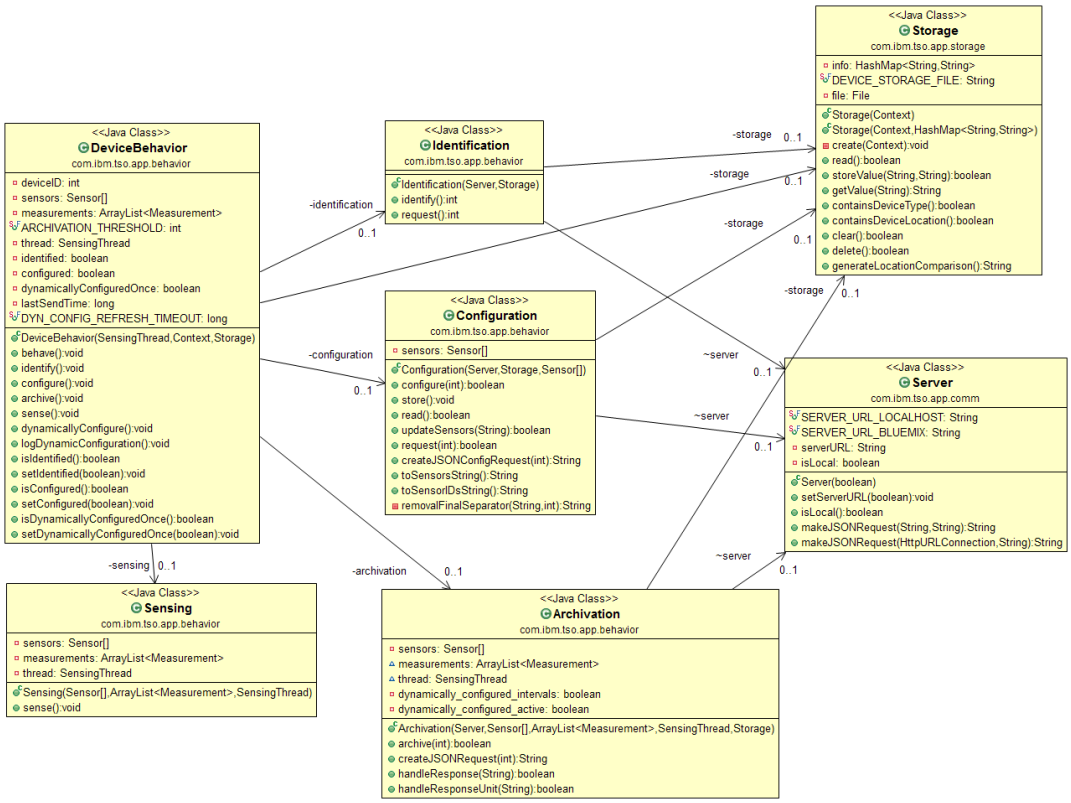

Figure 4.2: UML Class Diagram of the Sensor Node Behavior (generated using ObjectAid UML Explorer)

The most important design choices within the Android behavior implementation are as follows:

Operational Module Isolation The communication is completely done through the *Server* class, and the storage by the *Storage* class. This shelters the other classes from its internal workings, and enables abstraction on their side.

Phase Isolation Each of the phases described in the software design is directly translated into a single class, respectively the *Identification*, *Configuration*, *Archivation*, and *Sensing* classes. This allows for easy adaptability of the program flow in the *DeviceBehavior* class, and enables it to describe only a high level of program flow.

ANDROID INTERFACE IMPLEMENTATION

The interface is visually depicted using a UML diagram in figure 4.3. As can be seen, the *SensingThread* class is the connector between the behavior, represented by the *DeviceBehavior* class, and the interface, represented by the *MainActivity* class.

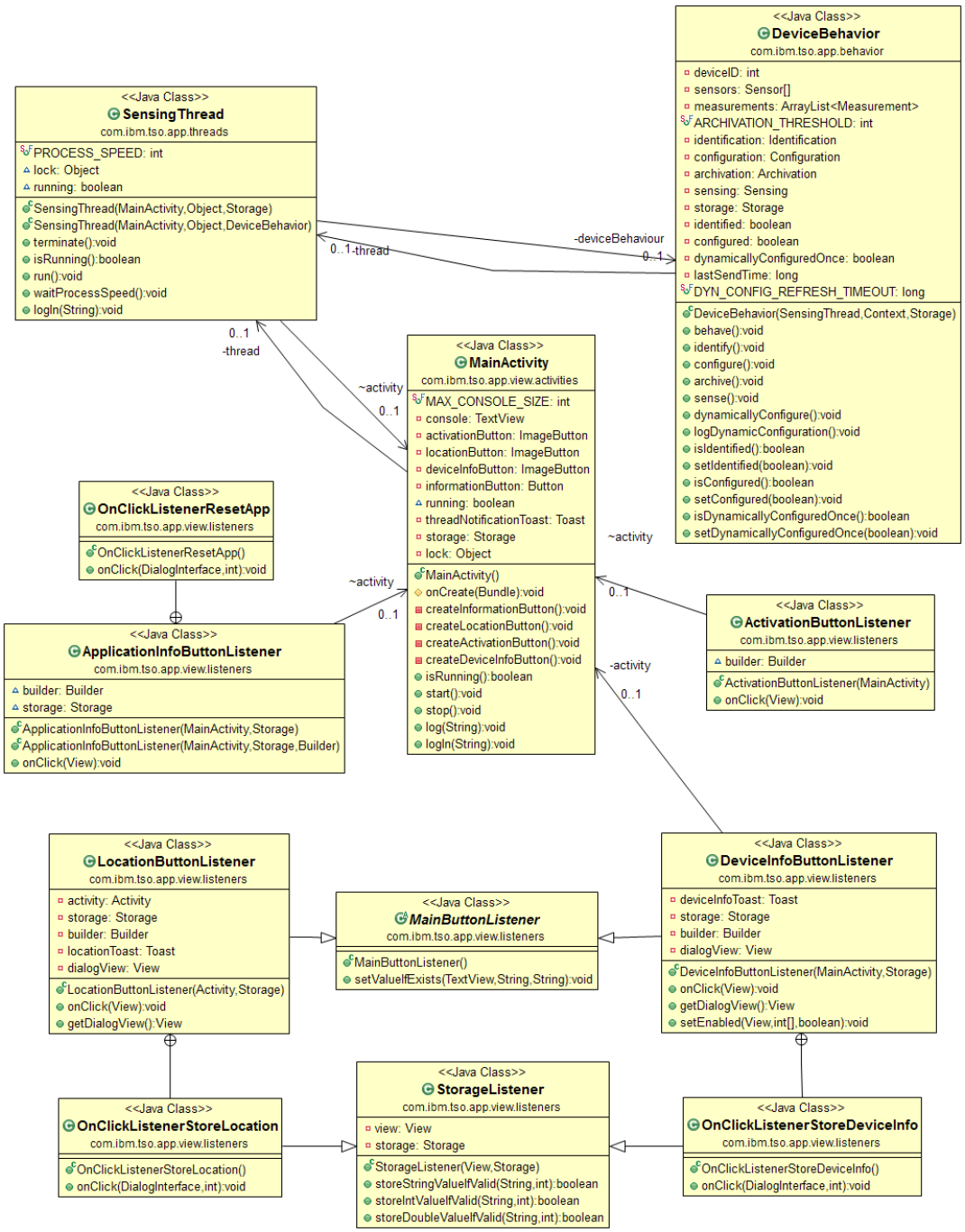

Figure 4.3: UML Class Diagram of the Sensor Node Interface (generated using ObjectAid UML Explorer)

The most important design choices within the Android interface implementation are as follows:

Single Activity Because the application is small, and only requires little user interaction, a single activity exists. This eliminates the need to transfer data between multiple dependent activities.

Thread Separation The behavior runs on another thread than the main activity. This enables the interface to remain responsive, even if the behavior is blocking (e.g. when a network request is performed). The

main thread is however still in full control, and can, through the *SensingThread* class, terminate the sensor node behavior at the finger tips of the user.

Listeners and Dialogs To separate the *MainActivity* class intialization task from the interface, separate listener classes are created for each of the UI elements. The listeners are responsible for the subsequent interaction with the user through a dialog.

ARDUINO IMPLEMENTATION

The class hierarchy of the C++ implementation is visually depicted in figure 4.4. The main module (which is not a class) follows almost literally the software design pattern laid out in section 4.2.3.

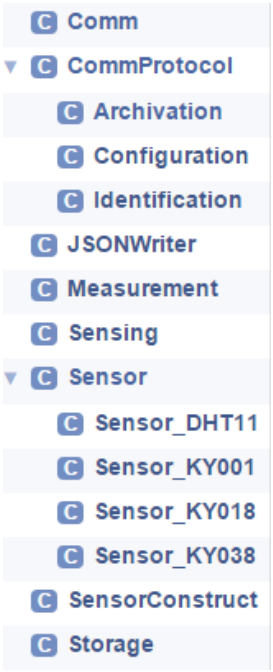

Figure 4.4: Class Hierarchy of the Arduino implementation (generated using Doxygen)

The most important design choices within the Arduino implementation are as follows:

Sensor Abstraction Each sensor type follows the abstraction provided by the *Sensor* class. Tasks that are ubiquitous, such as interval handling, are implemented in the parent class. More sensor-specific functionality, such as value measurement, are implemented by the respective sensor subclasses themselves.

Phase Isolation The four phases have been implemented separately, enabling abstraction in the main module. The *Identification*, *Configuration*, and *Archivation*, are all implementations for the *CommProtocol* abstract class. This simplifies the communication, as they only need to implement the request and the final handling of the response. All other communicative functionality is taken care of by the *Comm* class.

Operational Module Isolation The communication is completely done through the *Comm* class, and the storage by the *Storage* class. These needed to implemented at a particularly low-level. The *Comm* class needs to communicate via I/O sockets using a Serial Port Profile with the hub node, which requires a lot of timing. Furthermore, the *Storage* class needs to convert data types to bytes, which are mapped and stored in the EEPROM memory.

4.3. HUB NODE

The hub node component has been implemented on a single platform (Java) for the demonstrator, however, due to the selection of this specific (widely supported) platform, an enormous range of devices can be reached. In this section, the implementation of the hub node is discussed. First, in section 4.3.1, the chosen hardware and software dependencies for the hub node are explained. Second, in section 4.3.2, the software design is elaborated upon. Third, in section 4.3.3, the software implementation is discussed.

4.3.1. SOFTWARE DEPENDENCIES AND HARDWARE

At the inception of the project, the usage of dedicated (relatively low-cost) hub nodes was seen as the obvious choice. In the research phase, we performed testing on the Raspberry Pi platform. These experiments uncovered three major issues. First, it proved extremely challenging to connect these to the IBM network because of their status as unknown hardware (all hardware connected to the IBM network had to be registered before they could make a connection). The network administrator was unwilling to comply, as it would result in a major security drop, if they would allow unknown hardware to be connected to the network. Second, the Raspberry Pi does not natively support a Bluetooth stack, and thus requires an additional Bluetooth module. Due to time constraints put on the project, there was insufficient time to properly test and evaluate the Bluetooth modules. Third, debugging proved to be difficult, with the absence of HDMI-support at IBM. The support for the Raspberry Pi has however not been abandoned.

Instead of deploying dedicated hub nodes, in discussion with the client it was suggested to use existing infrastructure of laptops used by employees. These laptops possessed both a well-documented and well-supported protocol stack (WINSOCK), and Java. To maintain as much platform independence, a Java Bluetooth library was chosen that supported as many operating systems and protocols as possible: BlueCove[14]. With the support of Mac OS X, WIDCOMM, BlueSoleil and Microsoft Bluetooth (SP2 and later), it proved to be the best solution. Furthermore, the latest version of the JUnit framework was used for testing, due to its overwhelming adoption by Java developers, its vast documentation, and many additional powerful add-ons (such as PowerMock for mocking objects). The EclEmma tool is used to analyse the coverage of the JUnit test results, due to their excellent Eclipse plug-in and seamless integration with JUnit. An overview of the software dependencies is depicted in table 4.4.

Name	Functionality	License
BlueCove	Bluetooth protocol stack wrapper	Apache Software License v2.0
JUnit 4	Unit testing framework	Eclipse Public License v1.0
EclEmma	Coverage Tool	Eclipse Public License v1.0

Table 4.4: Hub node software dependencies

4.3.2. SOFTWARE DESIGN

This section described how the software is designed (using the dependencies in 4.2.2) from a high level point-of-view . In the first section, an overview is given of the operational modules on which the program flow depends. In the second section, the program flow of the main thread is discussed. In the third section, the program flow of the communication thread is elaborated upon.

OPERATIONAL MODULES

Throughout the hub node there are modules on which the main program flow relies. These can be implemented in various ways (both explicitly in a class, or implicitly in the program), but always offer the same functionality. There are four such modules:

Bluetooth The module responsible for communication with another Bluetooth device through the Serial Port Profile (SPP) protocol. It can detect devices, open and close connections, and read and write.

Server The module responsible for communication with the server. This is implemented with multithreading support, allowing for multiple requests to be handled simultaneously. It receives an endpoint and message from the caller, passes this via a POST request to the server, and then returns the response to the caller.

[14] The BlueCove project is hosted at: `https://code.google.com/p/bluecove/` (retrieved 10 June 2015)

Storage The module responsible for persistent storage of variables on the hub node. It can store and retrieve the identification.

Communication Threads List of started communication threads. Each communication thread tries to connect with the sensor node it was started for, if it fails, it will die. If it succeeds, it will continue to communicate with the sensor node.

HUB NODE MAIN THREAD

The main thread of the hub node is responsible for identifying itself with the server, and maintaining the connections with the sensor nodes. The connections themselves are managed by separate communication threads, as will be elaborated upon in the following section.

Listing 4.2: Main Thread Pseudo-code

```
input: bluetooth, server, storage, communicationthreads

begin
    while (true) {

        if not identified then
                identify()
        else then
                discover_devices()
                filter_sensor_nodes()
                communicate_with_unconnected_sensor_nodes()
                wait_for_next_discovery()

    }
end
```

The following phases are present in the main thread of the hub node:

identify() Checks whether an identification (i.e. device identifier) exists in the persistent storage. If it does not, it requests a new identification via the communication from the server (giving its device type and initial location) and stores it persistently.

discover_devices() Discover nearby (paired) Bluetooth devices.

filter_sensing_devices() Filter the list of discovered devices, looking for sensor nodes (this can either be done through their services or name).

communicate_with_unconnected_sensor_nodes() Start a separate communication thread for each of the sensor nodes that are not connected. Also removes threads that died, and starts new replacement threads in an attempt to reconnect.

wait_for_next_discovery() Discovery of devices is an intensive and blocking procedure, this phase waits a set time to look for devices again (allowing the actual communication to take place in the meanwhile).

HUB NODE COMMUNICATION THREAD

The communication threads of the hub node are responsible for maintaining the connection with the individual sensor nodes. Each thread attempts to connect to the sensor nodes in a timely fashion, and asks whether they have any request. If there are requests, the communication thread functions as a channel between sensor node and server.

Listing 4.3: Hub Node Communication Thread Pseudo-code

```
input: bluetooth, server

begin
    while (not dead) {

        if not attempt_connection() then
            die()

        send_start_signal()

        if received request then
            adapt_request()
            send_request_to_server()
            await_server_response()
            send_back_response_to_sensor_node()

        wait_for_next_attempt_connection()

    }
end
```

The following phases are present in the communication threads of the hub node:

attempt_connection() Attempt to connect with the device to which this thread is dedicated. If it fails, the thread will die. A new attempt is done for the next time when the main thread rediscovers the devices.

send_start_signal() Send the start signal to the sensor node. This is a predefined sequence that informs the sensor node the hub is ready to receive a request.

adapt_request() Slightly adapt the request, adding the hub identifier and the delay it induced.

send_request_to_server() Send the adapted request to the server.

await_server_response() Await the response of the server.

send_back_response_to_sensor_node() Send back the response it received from the server.

wait_for_next_attempt_connection() The hub node asks the device in an interval if it has any message. This phase is waiting for the interval to pass.

4.3.3. SOFTWARE IMPLEMENTATION

As explained in section 4.3.1, the software was implemented using Java. Below, the structure is briefly explained. For specific implementation inquiries, it is best to directly look at the code.

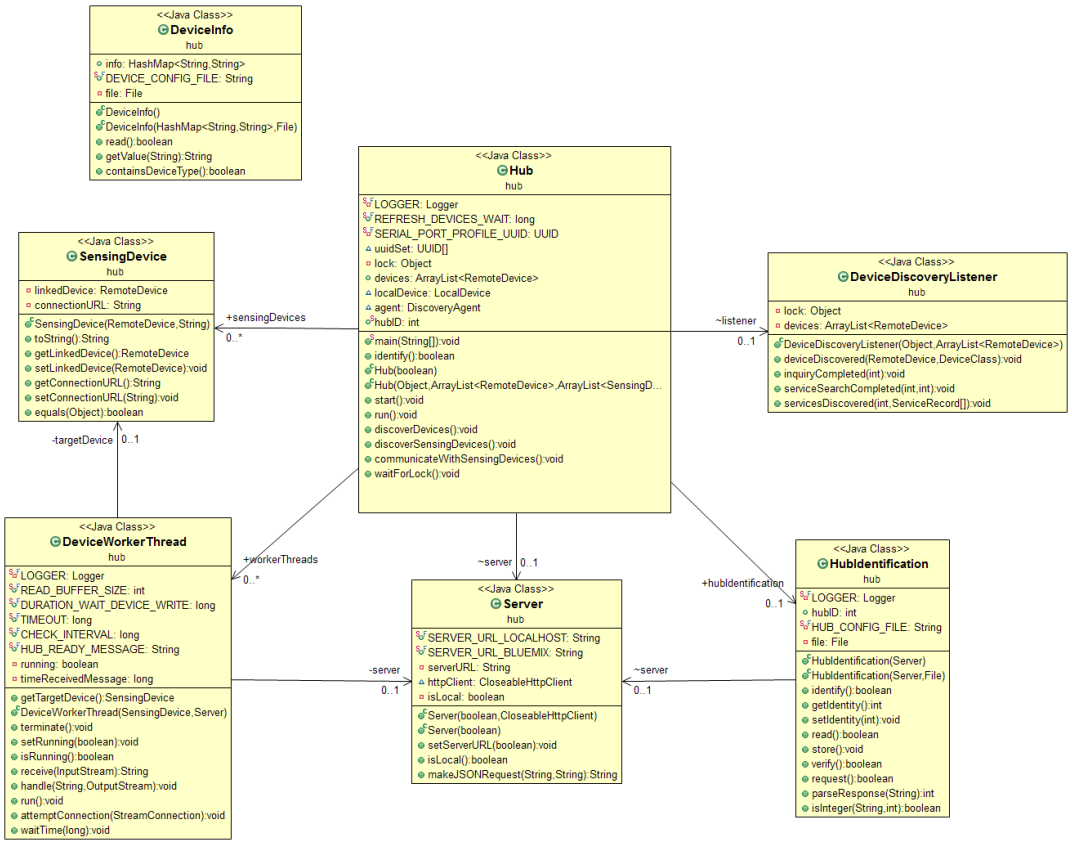

Figure 4.5: UML Class Diagram of the Hub Node

The most significant implementation choices are explained below:

Asynchronous Device Discovery The BlueCove library performs the device discovery on a separate thread. The *DeviceDiscoveryListener* class is created to handle its response and to notify the waiting 'locked' main thread of the *Hub* class.

Identification The *HubIdentification* class is entirely responsible for the request and storage of the hub identification. It relies on the *Server* class to handle the former.

Storage of Device Information The device information (e.g. name, type) is pulled from a configuration file upon the starting of the hub. The *DeviceInfo* class is fully responsible for the delivery of this information.

Multithreading The *Hub* class is run on the main thread. After discovery of sensor nodes, it creates additional *DeviceWorkerThreads* to handle the communication with each separate device. This enables support for Bluetooth stacks that allow simultaneous connections, and thus increases the scalability of the hub node.

4.4. SERVER

The server is implemented using NodeJS and was deployed with IBM Bluemix, which will be substantiated below. First, in section 4.4.1, the chosen hardware for the server - IBM Bluemix and DB2 - will be discussed. Second, in section 4.4.2, the chosen software - NodeJS - for the server will be elaborated upon, and its dependencies justified. Third, in section 4.4.3, the software design is explained. Finally, in section 4.4.4, the implementation is elaborated upon.

4.4.1. HARDWARE

In this section, the chosen hardware for the server will be discussed. Below the choice of using IBM Bluemix will firstly be elaborated upon, followed by a discussion regarding the usage of IBM's DB2.

BLUEMIX

IBM Bluemix was chosen to host both the server and the database. This choice was made based on five reasons:

Client's Wishes The client, IBM, has strong preference to use IBM technologies. This allows them to be in full control of the project, instead of relying on third party hosting. In addition, the usage of Bluemix was in our case free of charge as it considered for internal usage regarding research.

Private Git Repositories Through the DevOps programme of IBM, it is possible to host private Git repositories on Bluemix. This Git repository is then coupled to the Bluemix app, enabling smooth deployment integration.

Low Latency Database Connection Both the chosen database for the demonstrator, DB2, and the server, are hosted on Bluemix. This provides a reliable connection with little latency (as they are part of the same internal network).

Build and Deploy Stages Bluemix allows developers to stage releases, and, in case of a faulty release, revert to previous versions. This control enables developers to have full control over what is released.

Automatic Dependency Loading Bluemix reads the project's configuration, and will load in the modules it depends on. It supports multiple versions of several modules such as NodeJS. It uses the default NPM installer, thus ensures that when it works locally, it will also work when the system goes 'live'.

Extremely Scalable Bluemix allows developers to dynamically adjust the bandwidth and memory allocated for an application within a very large range. This allows the developers to increase capacity during stressful periods, and decrease in less pressing times.

IBM DB2 DATABASE

IBM's database, DB2, was chosen to be the database on which the server would rely. In early product development, PostGreSQL was used. Due to two reasons, a switch was made to IBM's product. The first reason was the insistence of the client, whom preferred the usage of an internal database. This would namely enable IBM to be in full control and the single possessor of the data. Secondly, as IBM hosts the DB2 database, the available space is unlimited. Given the expected size of the dataset, hosting by an external party could become quite expensive when the system is deployed. Apart from a few minor changes in the syntax, the two database types (PostGreSQL and DB2) are interchangeable. Database independence is further discussed in section 8.5.5.

4.4.2. SOFTWARE DEPENDENCIES

In this section, choices of software dependencies for the server are explained. First, the main system software dependencies are elaborated upon. These software dependencies are used in the actual production code. Second, the test software dependencies are explained. These dependencies are only used for debugging and testing the system, and are not part of the main system.

MAIN SYSTEM

NodeJS[15] was chosen as the programming language. The choice was motivated by five factors. First of all, NodeJS is an open-source project with an active community, offering wide support and regular updates. Second, NodeJS matches our need for a logic structure of endpoints. Third, its syntax is the same as JavaScript, to which many developers are accustomed, as it can be seen as the dominating scripting language on the Web at this moment. Fourth, the NodeJS Package Manager (NPM) enables a Maven-like control over dependencies. Fifth, NodeJS is supported by Bluemix. Alternatives that were also supported by Bluemix like Liberty for Java, and the .go, .php, .py, .ruby and .net languages, which did not have as much preference, as they offered less benefits.

Express, the "fast, unopinionated, minimalist web framework for Node.js"[16], was chosen for running the NodeJS server application. Alternatives[17] like the Koa[18] and Hapi[19] frameworks are, next to being less popular and thus having a smaller community, respectively still under heavy development and more aimed at creating larger applications. The latter would result in a considerable amount of boilerplate code for creating a server with our (considerably small) needs.

Next to the web framework, a parser needed to be integrated to parse the JSON requests. For this purpose, we included the Body Parser parsing middleware, which can be seen as the defacto standard regarding parsing in combination with Express. As will be explained in section 8.5, in future development this JSON parser can be removed when using self created requests, enabling for even more efficient communication by lowering the message length. Though, as the demonstrator still is and will be under development, we chose to use JSON requests as it allows for a more visualizable communication in development. This is the result of JSON's descriptive structure, and usage of operators well-known under programmers. The IBM DB module was added to communicate with the DB2 database. To date, this is the only module that supports the DB2 database communication, which bound us to usage of this module. An overview of the described system software dependencies can be found in table 4.5.

Name	Functionality	License
Express	Web framework	MIT License
Body Parser	Parsing of request into JSON	MIT License
IBM DB	Connection client for DB2 database	MIT License

Table 4.5: Server main software dependencies

TESTING

After choosing NodeJS for implementing the server and IBM DB for accessing the DB2 database, it was required to choose a testing toolkit, starting with a testing framework. For this, we chose Mocha[20], "the gold standard" and a "well tested and maintained"[21] framework for testing with NodeJS. Next to this framework, we also included Istanbul. Istanbul is a coverage tool for JavaScript, and has standard support for Mocha. For comprehensability of logging messages, the colors[22] module was added as well, which allows for coloring the console output. A separate assertion library was not needed (e.g. Assert, Chai or should.js), because NodeJS's own assert module already provided us with the needed functionality. An overview of the described testing software dependencies is given in table 4.6.

[15] https://nodejs.org/
[16] http://expressjs.com/
[17] A comparison done by Jonathan Glock can be found on:
https://www.airpair.com/node.js/posts/nodejs-framework-comparison-express-koa-hapi (retrieved 18 June 2015)
[18] http://koajs.com/
[19] http://hapijs.com/
[20] http://mochajs.org/
[21] http://thenodeway.io/posts/testing-essentials/
[22] https://github.com/Marak/colors.js

Name	Functionality	License
Mocha	Asynchronous testing framework	MIT License
Istanbul	Coverage	BSD-3-Clause license
Colors	Colouring of console messages	MIT License

Table 4.6: Server testing software dependencies

4.4.3. SOFTWARE DESIGN

The NodeJS structure inherently supports modularity, as is attested by the effort to uncouple so-called combination packages (e.g. early versions of Express) into separate packages (e.g. Express, Body Parser, JSON Parser). The server implements four endpoints to fulfill its responsibilities laid out in section 3.2.4. These four endpoints share a demand for certain functionality, namely database communication (e.g. insertion, deletion) and certain utilities (e.g. logging, validation). First, this highly demanded functionality is discussed in the first two sections. Second, the endpoints are described in the following four sections.

DATABASE QUERY MODULE

All throughout the server, a database connection is used. The database query module acts as an interface between the server and the database. It offers semantic functionality, and abstracts over the SQL queries. This enables the server to become database independent, which would allow for other databases to be used in future stages of the product. The database query module is separated into eight sub-modules, which have dedicated responsibilities:

insert Blind insertion of new entities into the database.

select Selection of entities of the database.

remove Removal of entities from the database.

check Checking the validity of identifiers, that supposedly match entities in the database.

forced-get Retrieval of identifiers of the database. The functions also receives additional parameters, that are used when the identifier could not be found to create a new entity.

update Updating entities of the database.

date-conv Conversion between the date objects of JavaScript and the date standards of the database.

sys-defaults Maintaining the settings of the system, such as a dynamic configuration interval for sensor nodes.

UTILITY MODULE

The utility module contains functionality that all endpoint handlers use, but does not belong to the category of 'database functionality'. There are two sub-modules:

validator Validates the input variables the endpoints receive in the bodies of requests.

logm Manager for the logging of messages. It supports color-coding, which drastically improves reading comprehensibility.

HUB NODE IDENTIFICATION ENDPOINT

Each hub has to identify itself before relaying any communication between sensor nodes and the server. This endpoint handles that identification. The request itself contains device information, i.e. the chosen name and its device type. There are two main flows, if an hub identifier is already given, and if not. The former requires merely a check, whereas the latter requires the creation of an entirely new hub.

Listing 4.4: Hub Identification Pseudo-code

```
input: hub_id, name, device_type

begin
    read_input_from_body()

    if hub_id is given then
        if not check(hub_id) then
            end FAILURE

    else then
        if not valid(device_type) then
            end FAILURE

        get_device_type_id()

        insert_new_hub()

    end SUCCESS

end
```

SENSOR NODE IDENTIFICATION ENDPOINT

Each sensor node has to identify itself with the server before either configuration or submission of measurements can take place. This endpoint handles the identification. The request itself contains an existing identifier (optional), name, device type, and its current location. The server checks whether the identifier is valid, or creates and returns a new one. Next, it checks whether a location exists. If it does not, it creates and returns a new location. If it does, it will update the database with the most up-to-date variant.

Listing 4.5: Device Identification Pseudo-code

```
input: device_id, name, device_type, location

begin
    read_input_from_body()

    if device_id is given then
        if not check(device_id) then
            end FAILURE

    else then
        if not valid(device_type) then
            end FAILURE

        get_device_type_id()

        insert_new_device()

    location_id := NULL

    if location is not given then
        location_id = select_latest_location()
    else
        if valid(location) and check(location.environment)
            location_id = get_location_id(location)


    if not valid(location_id) then
        end FAILURE

    end SUCCESS

end
```

SENSOR NODE CONFIGURATION ENDPOINT

After a sensor node has identified itself with the server, it needs to configure its static configuration (sensors) with the server. This endpoint handles that configuration. The request contains the device identification, and an array of the sensors that are attached to the sensor node. The sensor node can configure itself multiple times, e.g. when a sensor has been removed. Hence, a time conflict check is added. If the sensors are valid, it will then iterate over each sensor, and insert each sensor as an entity into the database. The resulting sensor identifiers are returned in the response.

Listing 4.6: Device Configuration Pseudo-code

```
input: device_id, sensors

begin
    read_input_from_body()

    if device_id is given then
        if not check(device_id) then
            end FAILURE

        if select(latest_config) and time conflicts then
            end FAILURE

        insert_new_device_configuration()

        if valid(sensors) then
            for each sensor in sensors:
                if valid(sensor)

                    get_sensor_type()
                    insert_sensor()
                    insert_sensor_activity()


        end SUCCESS

    end FAILURE

end
```

SENSOR NODE MEASUREMENTS ENDPOINT

The sensor nodes regularly submit their measurements to the server. This regular interaction is used to support the dynamic configuration as well. The measurements endpoint firstly checks whether all the input it receives is valid. The sensor and hub node obtained most of the information from the identification and configuration with the other endpoints. Second, the endpoint makes sure that a new location is stored, if one is given. Third, it iterates over all the measurements and inserts them, if they are valid. Finally, it checks whether the sensor node needs to receive an update on the dynamic configuration it should follow.

Listing 4.7: Measurements Pseudo-code

```
input: device_id, hub_id, hub_delay, location, measurements

begin
    read_input_from_body()

    read_time()

    if not valid(device_id, measurements)
        end FAILURE

    if not check(device_id)
        end FAILURE

    if exists hub_id and not check(hub_id) then
        end FAILURE

    location_id = get_location_id(location);

    if location_id is NULL then
        end FAILURE

    for each measurement in measurements:
        if valid(measurement)
            insert_measurement(location_id, hub_id, hub_delay, time, measurement)

    update_dynamic_configuration()

    end SUCCESS

end
```

4.4.4. SOFTWARE IMPLEMENTATION

The previously described software design has been implemented in their respective four modules. As can be derived from the naming of certain high level functions, the main handler modules depend on a set of database and utility modules. A module diagram of the server is shown in figure 4.6.

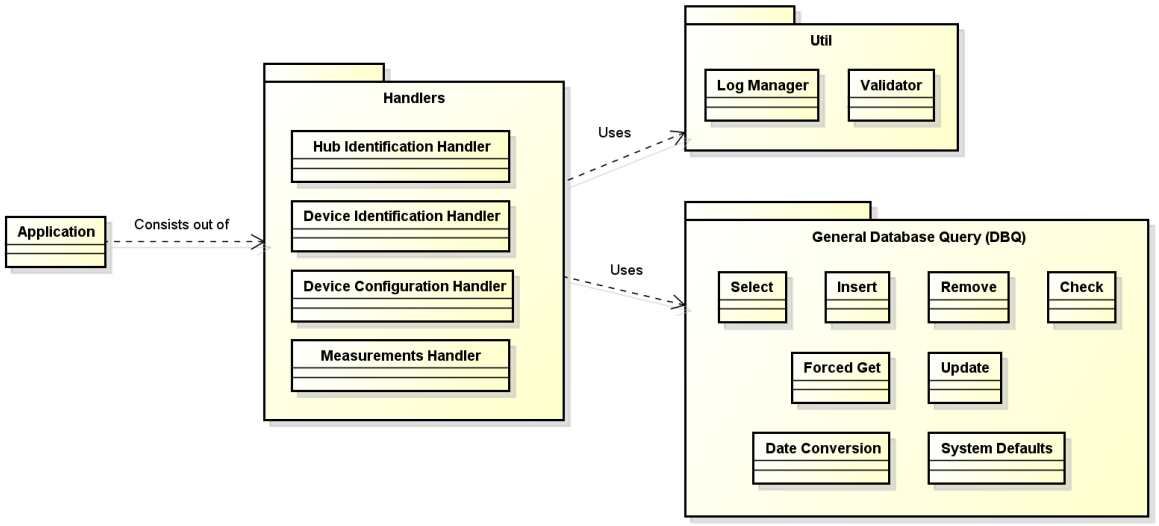

Figure 4.6: Module diagram for the server

The most important implementation choices are discussed below:

- **Database Isolation:** The main module handlers do not directly communicate with the database through SQL queries, but rather make use of the DBQ modules. This allows the handlers to abstract from the database chosen.
- **Logging Control:** Logging on the server is essential for debugging purposes. The logging manager allows abstraction from where (e.g. in the console, or in a file) and how (e.g. colouring) the logging messages are written.

4.5. DATABASE

The Entity Relationship model that was designed in section 3.3.1 has been implemented in a relational scheme. In section 4.5.1 the relational scheme diagram is depicted. Each unit of the relational scheme, a *relation* or *table*, is described in the following sections. This includes both the constraints it leaves for the system (including which component is responsible for it) and possible improvements. It respectively discusses the SensorType (4.5.2), DeviceType (4.5.3), Environment (4.5.4), SensingDevice ((4.5.5), SensingDeviceConfiguration (4.5.6), Sensor (4.5.3), SensorActivity (4.5.8), Hub (4.5.9), SensingDeviceLocation (4.5.10), and Measurement (4.5.11) tables.

4.5.1. RELATIONAL SCHEME

The translation from a relational scheme to relational database is done through Structured Query Language (SQL) queries. In appendix C and D the SQL queries for respectively the PostGreSQL and IBM DB2 databases are enumerated. Both databases have been used in the development, although in the final iterations and deployment, IBM DB2 was used (per request of the client). An overview of the relational scheme is depicted in figure 4.7.

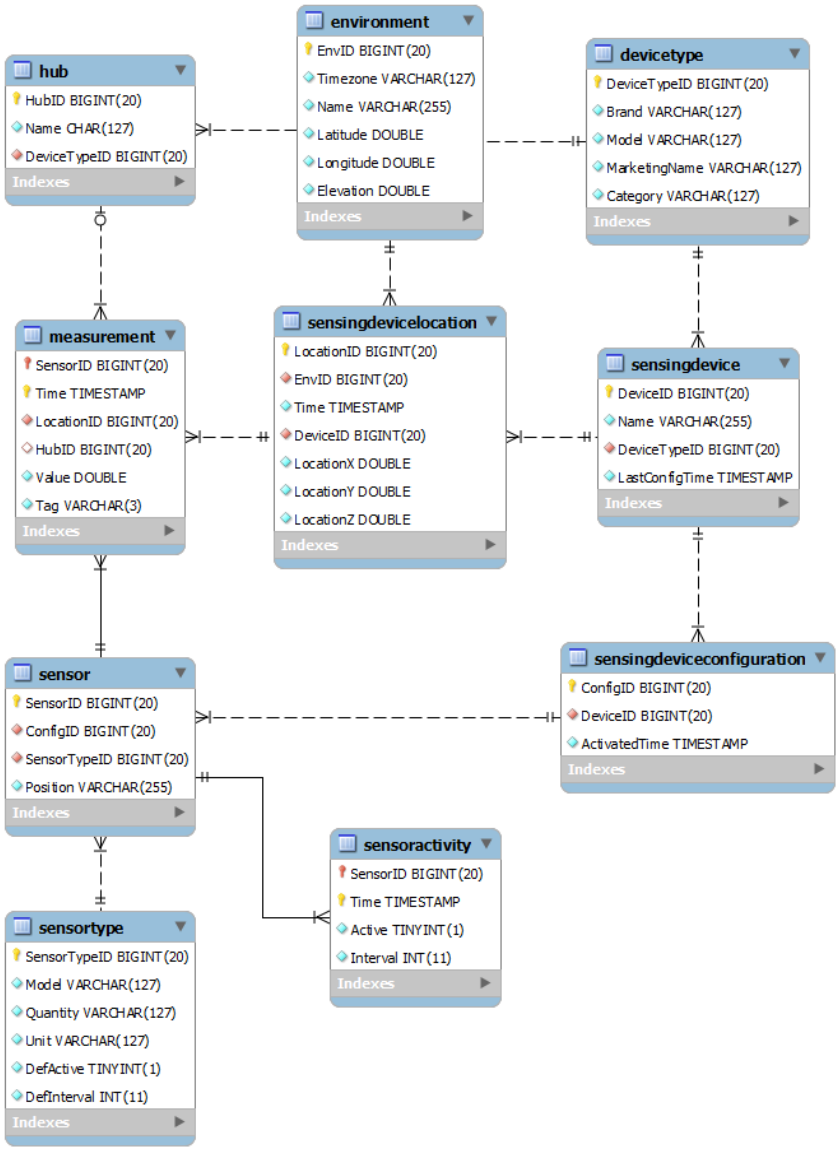

Figure 4.7: Relational Scheme Diagram (generated using MySQL Workbench)

4.5.2. SENSORTYPE TABLE

Directly translated from the *SensorType* entity. To minimize communication with devices, the *SensorTypeID* is set as primary key for the table. The combination of Model and Quantity is marked as unique because a single sensor type can measure multiple quantities. Units of the same quantity are translatable into each other (e.g. Celsius and Fahrenheit), but are stored in the quantity of the actual measurement for comprehensibility. The default activity and interval are used in the creation of the first sensor activity for a newly created sensor.

Constraint(s) left to system: the devices are responsible for providing proper model names and quantities. The server does not check the validity of these received sensor types. A recommendation is to uppercase all model and quantity values.

Possible improvements: extra information about the sensor types could be added, such as precision, or scale.

Column	Description	Data type	Special attributes
SensorTypeID	Unique identifier of the device (assigned)	SERIAL/INT	Primary key, auto increment
Model	Sensor model (e.g. “KY001”)	VARCHAR(127)	Part of unique combination with Quantity
Quantity	Physics Quantity (e.g. “TEMPERATURE)	VARCHAR(127)	Part of unique combination with Model
Unit	Unit of measurement (e.g. “CELSIUS”)	VARCHAR(127)	/
DefActive	Default activity (e.g. true)	BOOL/SMALLINT	/
DefInterval	Default interval in ms (e.g. 5000)	INT	/

Table 4.7: SensorType table (SQL in C.2 and D.2)

4.5.3. DEVICETYPE TABLE

Directly translated from the DeviceType entity. Because devices have to identify themselves with each communication, a single column *DeviceID* is added and as primary key for the table. The combination of *Brand* and *Model* is set as unique. Marketing name and category are mere descriptive columns.

Constraint(s) left to system: the devices are responsible for providing a proper brand, model, and marketing name. For the low-cost devices, this requires providing the cloud application with a hard-coded brand, model and name, but for the other sensor nodes (e.g. smartphone) we expect that this can be deduced by the application.

Possible improvements: link this table with external databases containing information about hardware instead of relying on devices to provide accurate information.

Column	Description	Data type	Special attributes
DeviceTypeID	Unique identifier of the device type (assigned)	SERIAL/INT	Primary key, auto increment
Brand	Brand of device (e.g. “Samsung”)	VARCHAR(127)	Part of unique combination with Model
Model	Model of device (e.g. “SHV-3958”)	VARCHAR(127)	Part of unique combination with Brand
MarketingName	Marketing name (e.g. “Galaxy S3”)	VARCHAR(127)	/
Category	Category it belongs to (e.g. "smartphone’)	VARCHAR(127)	/

Table 4.8: DeviceType table (SQL in C.3 and D.3)

4.5.4. ENVIRONMENT TABLE

Directly translated from the *Environment* entity. *EnvID* is added as primary key, this has two reasons: (1) this table is referenced by every tuple of *SensingDeviceLocation*, highlighting a small primary key, and (2) there is not really a different candidate key than a uniquely chosen Name. The location of the environment is determined globally by *Latitude*, *Longitude*, and *Elevation*. To localize the time of the measurements, the *Timezone* column is added.

Constraint(s) left to system: the sensor nodes themselves are responsible for detecting the environment they are in. Either through beacons, or through hard-coded values.

Possible improvements: determine what makes an environment unique, instead of solely relying on the generated identifier. This is difficult, because it is not yet known where the environments will be. Identifying information such as "Company" might not be applicable in all situations.

Column	Description	Data type	Special attributes
EnvID	Unique identifier of the environment	SERIAL/INT	Primary key, auto increment
Timezone	Timezone (e.g. GMT+1)	CHAR(127)	/
Name	Name given (e.g. "IBM CAS")	CHAR(255)	/
Latitude	GPS Latitude	DOUBLE PRECISION	/
Longitude	GPS Longitude	DOUBLE PRECISION	/
Elevation	Elevation	DOUBLE PRECISION	/

Table 4.9: Environment table (SQL in C.4 and D.4)

4.5.5. SENSINGDEVICE TABLE

Directly translated from the *SensingDevice* entity. The Name column is provided by the device. All fields are mandatory. To prevent manually assigning identifiers to devices, the *DeviceID* is automatically generated.

Constraint(s) left to system: the name of the device should be chosen to be identifiable for humans. The server is responsible for updating the last configuration time.

Possible improvements: addition of a descriptive column for the operating system, which is being run on the device.

Column	Description	Data type	Special attributes
DeviceID	Unique identifier of the device (assigned)	SERIAL/INT	Primary key, auto increment
Name	Given name of the device (given by device)	VARCHAR(255)	/
LastConfigTime	Last time the device was dynamically configured	TIMESTAMP	/
DeviceTypeID	Identifier to the type of device	INT	References DeviceType.DeviceTypeID

Table 4.10: SensingDevice table (SQL in C.5 and D.5)

4.5.6. SENSINGDEVICECONFIGURATION TABLE

Directly translated from the *SensingDeviceConfiguration* entity. To ensure that no device can have two configurations at the same time, columns *DeviceID* and *ActivatedTime* form a unique combination. The *ConfigID* column is added to allow the coupling with sensors (and ultimately measurements) without having to copy the unique combination of *DeviceID* and *ActivatedTime*, which would require considerably more space. This unique constraint forces the creation of an index, which can be used to quickly retrieve the current sensing configuration of a device.

Constraint(s) left to system: there must always be a device configuration for a sensing device. Before measurements are sent, the device must have passed on its configuration.

Column	Description	Data type	Special attributes
ConfigID	Unique identifier for the device configuration	SERIAL/INT	Primary key, auto increment
DeviceID	Unique identifier of the device (assigned)	INT	Part of unique combination with ActivatedTime, references SensingDevice.DeviceID
ActivatedTime	When the configuration was activated	TIMESTAMP	Part of unique combination with DeviceID

Table 4.11: SensingDeviceConfiguration table (SQL in C.6 and D.6)

4.5.7. SENSOR TABLE

Directly translated from the *Sensor* entity. To ensure that a device only has differently named sensors in a specific configuration, columns *ConfigID*, *SensorTypeID*, and *Name* form a unique combination. The *SensorID* column is added to prevent the copy of *ConfigID*, *SensorTypeID*, and *Name* into the measurements to uniquely identify them, which would require significantly more space considering the vast amount of measurements to be made in this system.

Constraint(s) left to system: the devices are responsible to give names to the sensors in case there are multiple of the same type. This can happen when the sensing device measures light in five directions, which could have the names “NORTH”, “EAST”, “SOUTH”, “WEST”, “TOP”. It can also be more subjective, e.g. “TOWARDS PERSON”, “TOWARDS LIGHTS”.

Column	Description	Data type	Special attributes
SensorID	Unique identifier for the sensor	INT	Primary key, auto increment
ConfigID	Configuration identifier in which the sensor is present	INT	References DeviceConfiguration.DeviceID,part of unique combination with SensorTypeID and Name
SensorTypeID	Sensor type identifier of the type the sensor is	INT	References SensorType.SensorTypeID,part of unique combination with ConfigID and Name
Position	Position name of the sensor on the device (e.g. "NORTH")	VARCHAR(255)	Part of unique combination with ConfigID and SensorTypeID

Table 4.12: Sensor table (SQL in C.7 and D.7)

4.5.8. SENSORACTIVITY TABLE

Directly translated from the *SensorActivity* entity. The column SensorID and Time form a unique combination.

Constraint(s) left to system: when a device configuration is created, for each sensor an entry must be made into this activity table (in the same transaction). This ensures that the database always knows whether a sensor is active or not.

Column	Description	Data type	Special attributes
SensorID	Unique identifier for the sensor	INT	Primary key
Time	Time at which the activity is changed	TIMESTAMP	Primary key
Active	Whether the sensor is now active	BOOLEAN	/
Interval	The interval at which the sensor measures	BOOLEAN	/

Table 4.13: SensorActivity table (SQL in C.8 and D.8)

4.5.9. HUB TABLE

Directly translated from the *Hub* entity. The column HubID is added so that it is not necessary to manually distribute identifiers amongst devices.

Constraint(s) left to system: the name should be chosen as identifiable as possible by the hub (e.g. "John Jameson's laptop").

Column	Description	Data type	Special attributes
HubID	Unique identifier for the hub	SERIAL	Primary key
Name	Given name of the hub	CHAR(127)	/
DeviceTypeID	Identifier of the type of device	INT	References DeviceType.DeviceTypeID

Table 4.14: Hub table (SQL in C.9 and D.9)

4.5.10. SENSINGDEVICELOCATION TABLE

Directly translated from the *SensingDeviceLocation* entity. This table is completely responsible for linking the location of devices to measurements. Because the location is referenced by each measurement tuple, the column *LocationID* is added to conserve space. The columns *DeviceID* and *Time* form a unique combination. This unique combination forces the creation of an index that can be used to quickly retrieve the sensing device's last location.

Constraint(s) left to system: when a sensing device is created, an entry must be made into this table so that the server always knows a location for a device. The server is responsible for providing proper timestamps for the *Time* column.

Column	Description	Data type	Special attributes
LocationID	Unique identifier for this location	SERIAL/INT	Primary key
DeviceID	Identifier to the sensing device	INT	Part of unique combination with Time, references SensingDevice.DeviceID
Time	The time at which the location was registered	TIMESTAMP	Part of unique combination with DeviceID
EnvID	Environment identifier	INT	References Environment.EnvID
LocationX	X-coordinate relative to environment origin	REAL	/
LocationY	Y-coordinate relative to environment origin	REAL	/
LocationZ	Z-coordinate relative to environment origin	REAL	/

Table 4.15: SensingDeviceLocation table (SQL in C.10 and D.10)

4.5.11. MEASUREMENT TABLE

Directly translated from the *Measurement* entity. To ensure that a sensor can only produce one measurement at a time, the columns *SensorID* and *Time* form a unique combination. The location is determined with the *LocationID* column. A hub that might have passed on the measurement is registered via the *HubID* column.

Constraint(s) left to system: the server is responsible for providing proper timestamps for the *Time* column.

Column	Description	Data type	Special attributes
SensorID	Unique identifier of the sensor that measured	INT	References Sensor.SensorID,Part of unique combination with Time
Time	When the measurement was taken	TIMESTAMP	Part of unique combination with SensorID
LocationID	Identifier to the location it was taken	INT	References SensingDeviceLocation.LocationID
HubID	Identifier of the hub that aggregated it	INT	NULL value is allowed
Value	Value of the measurement	DOUBLE PRECISION	/
Tag	Tag of the measurement (e.g. "NOW", "MAX")	VARCHAR(3)	/

Table 4.16: Measurement table (SQL in C.11 and D.11)

5

TESTING

5.1. INTRODUCTION

This chapter is devoted to the testing of the system. First, the test plan is explained in section 5.2. Second, the unit testing for each of the system components is described in section 5.3. Third, in section 5.4, the testing of the integration of components is discussed. Finally, in section 5.5, the results of the demonstrator deployment are presented.

5.2. TEST PLAN

Similar to iterative development, testing is essential for a healthy software development procedure. It is a form of quality assurance: it points out flaws and defects in the system, it can evaluate performance, greatly increases maintainability of the system, and moreover, when done correctly, gives a sense of trust in the validity of the solution. As explained in section 5.2.1, a set of guidelines was composed to induce testing behavior. Three types of testing were performed in the product development of this system: Unit testing (5.2.2), Integration Testing (5.2.3), and System Testing (5.2.4).

5.2.1. TESTING GUIDELINES

To ensure proper testing throughout the development cycle, we set up a set of guidelines for ourselves to ensure product quality. The guidelines are as follows:

- Before committing to a local branch, the automated (unit) tests for each changed component must have been run and succeeded.
- Before merging a branch with the main repository, all automated (unit) tests for all components must have been run and succeeded. This supports that the main branch only contains working versions.
- Before labeling a commit in the main branch as a stable version, all automated (unit) tests, and integration tests must have been run and succeeded.

5.2.2. UNIT TESTING

Unit tests are performed on the component level. First, the unit tests for the least dependent classes are written, moving upward in the hierarchy tree. These tests are typically completely automated. In the case of interfaces, user interaction is mocked. A unit test class typically corresponds to a single class in the implementation, and tests all functionality of that class. Unit testing is very effective for regression testing, as it automatically tests all units within a system component (thus classifying it as 'still working' or 'broken'). All the unit testing frameworks and coverage tools mentioned in chapter 4 are added to an overview in table 5.1, for the convenience of the reader.

System Component	Unit Test Framework	Coverage Tool
Android Sensor Node	JUnit 3	EMMA
Arduino Sensor Node	ArduinoUnit	/
Hub Node	JUnit 4	EclEmma
Server	Mocha	Istanbul

Table 5.1: Unit Test Frameworks and Coverage Tools

5.2.3. INTEGRATION TESTING

Integration testing takes place between the different system components, but also between sub-components in a system component. Unlike unit testing, these tests can be semi-automated and may thus require manual intervention. The integration tests always take place in a developer-controlled environment. The following five integration tests were performed:

- **Smartphone UI:** exhaustively interact with the smartphone application's functionality through the user interface (UI). This includes enabling and disabling long range communication capabilities.
- **Smartphone Data Persistence:** run the server and activate the sensor node on the smartphone. Subsequently match the logging information shown in the application with the data in the database.
- **Component Dry-run:** running each system component without any other components active. Test their autonomous capabilities, and the absence of unexpected behavior due to dependence on other system components.
- **Arduino to Unconnected Hub:** test that the Arduino does not show unexpected behavior when connected to a Hub, which is not connected to the server.
- **Arduino Data Persistence:** run the server and hub, and activate the Arduino. Subsequently match the logging information produced by the Arduino with the data in the database.

5.2.4. SYSTEM TESTING

Deployment testing is performed at the end of the development cycle. It involves unleashing the system into the environment it was designed for. In the case of this demonstrator, the office of IBM CAS in Amsterdam. This is the final test of the system, as it involves all system components running together in a distributed fashion.

5.3. UNIT TESTING RESULTS

5.3.1. ANDROID SENSOR NODE

The *JUnit 3* framework was used to test the Android application (see section 4.2.2). To separate the test code from the main code, an additional Android project is created that depends on the main project. For each class (e.g. "NameOfClass"), a test class was made (e.g. "NameOfClassTest") following the same package structure, according to Android naming conventions[1]. A total of 17 test classes were made, that contain a total of 95 separate unit tests. The tests have been run on two devices, the Google Nexus 5 and Samsung Galaxy S5. The coverage statistics are displayed in table 5.2.

Package	Class	Method	Block	Line
com.ibm.tso.app.view.activities	100% (1/1)	100% (11/11)	70% (183/263)	66% (40.5/61)
com.ibm.tso.app.storage	100% (1/1)	100% (11/11)	81% (215/267)	78% (50.4/65)
com.ibm.tso.app.comm	100% (1/1)	80% (4/5)	82% (108/132)	89% (31/35)
com.ibm.tso.app.threads	100% (2/2)	100% (9/9)	93% (107/115)	94% (34.7/37)
com.ibm.tso.app.util	100% (1/1)	90% (9/10)	98% (218/223)	97% (44.8/46)
com.ibm.tso.app.view.listeners	100% (9/9)	100% (30/30)	98% (713/726)	98% (134.8/138)
com.ibm.tso.app.behavior	100% (5/5)	100% (34/34)	100% (1259/1261)	100% (230/231)
com.ibm.tso.app.sensors	100% (3/3)	100% (22/22)	100% (150/150)	100% (47/47)
Overall	**100% (23/23)**	**98% (130/132)**	**94% (2953/3137)**	**93% (613.2/660)**

Table 5.2: Coverage Statics for Android Unit Tests (generated by EMMA)

As can be seen in table 5.2, only two methods are not covered: one (in *comm*) that created an actual network socket, making it untestable, the second (in *util*) being the default constructor of an abstract class. There is a significant drop in the block and line coverage for the *activities* and *storage* package. The former is due to complications in testing multithreaded processes. The latter is caused by difficulties to trigger I/O exceptions when reading and writing files. Overall can be substantiated that, based on the statistics, the Android code is sufficiently covered. An overall coverage summary underlines this, stating a 100% class, 98% method, 94% block, and 93% line coverage according to EMMA.

5.3.2. ARDUINO SENSOR NODE

The *ArduinoUnit* framework was used to test the Arduino functionality (see section 4.2.2). Because the Arduino boards are embedded systems, the code requires platform-specific functionality to be executed. The code can thus only be executed on the Arduino devices themselves, which do not offer any coverage tools to date. Despite the absence of an objective coverage measurement, the code has been well-covered. For each class, a test class was made, which covered as many branches and lines of the respective class as possible. A total of 9 test classes were written, that cover the 13 classes in the main code. An overview is depicted in table 5.3. The proportional test size is given as an indication of how extensive testing was done.

Test Class (#lines)	Target Class(es) (#lines)	Proportional test size (%)
CommTest (410)	Comm (439)	93%
StorageTest (153)	Storage (192)	80%
SensorTest (143)	Sensor (154), Sensor_DHT11 (71), Sensor_KY001 (63), Sensor_KY018 (55), Sensor_KY038 (57)	36%[2]
SensingTest (172)	Sensing (102)	168%
JSONWriterTest (194)	JSONWriter (114)	170%
IdentificationTest (112)	Identification (91), CommProtocol (81/3)	95%
ConfigurationTest (127)	Configuration (119), CommProtocol (81/3)	87%
ArchivationTest (96)	Archivation (95), CommProtocol (81/3)	79%
Total Test Size: 1407	**Total Target Size: 1631**	**86%**

Table 5.3: Arduino Unit Tests Overview[3]

[1] http://developer.android.com/training/activity-testing/activity-basic-testing.html

5.3.3. HUB NODE

The *JUnit 4* framework was used to test the Java application for the hub node (see section 4.3.1). The project was a Maven2 project, and thus employed dynamic dependency management. Test naming and package structure conventions were upheld. Furthermore, a separation was made between the main code (*src/main/-java*) and test code (*src/test/java*). A total of 8 test classes were made, that contain a total of 48 separate unit tests. The tests have been run on Java version 1.7 and 1.8. The coverage statistics are displayed in figure 5.1.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
hub	90.7 %	2,709	279	2,988
src/main/java	78.3 %	988	274	1,262
hub	78.3 %	988	274	1,262
Hub.java	41.3 %	143	203	346
DeviceInfo.java	78.6 %	92	25	117
DeviceDiscoveryListener.java	90.6 %	29	3	32
DeviceWorkerThread.java	91.0 %	242	24	266
HubIdentification.java	94.4 %	319	19	338
SensingDevice.java	100.0 %	67	0	67
Server.java	100.0 %	96	0	96
src/test/java	99.7 %	1,721	5	1,726

Figure 5.1: Hub Coverage Report (generated by EclEmma)

Two classes stand out for their low coverage: *Hub* and *DeviceInfo*. The former is caused by its multithreading behavior. A possible solution could be redefining the threading structure, or using static code analysis. The latter, *DeviceInfo*, is caused by difficulties to trigger I/O exceptions in non-mockable objects. Overall can be substantiated, based on the statistics, that there is sufficient coverage, with an overall coverage of 78.3%.

5.3.4. SERVER

The *Mocha* framework was used to test the NodeJS application for the server (see section 4.4.2). The project was NodeJS project, and thus used the Node Package Manager (NPM) for dynamic dependency management. A clear separation was made in the main code and test code (only contained in folder *test*). By adding different scripts to the project package, multiple running test ways could be used. A total of 15 test classes were made. Moreover, a test database was created to separate the actual data from the testing, simultaneous testing and deployment. The coverage statistics are displayed in figure 5.2.

File	Statements		Branches		Functions		Lines	
dbq/	100%	(276 / 276)	88.64%	(39 / 44)	100%	(54 / 54)	100%	(276 / 276)
handlers/	97.06%	(132 / 136)	83.82%	(57 / 68)	100%	(7 / 7)	97.06%	(132 / 136)
util/	68.89%	(31 / 45)	54.17%	(13 / 24)	69.23%	(9 / 13)	68.89%	(31 / 45)

Figure 5.2: Server Coverage Report (generated by Istanbul)

As shown in figure 5.2, the database query (*dbq/*) and handler (*handlers/*) modules are well covered, scoring very high in statement, function and line coverage, and high in branch coverage. The are only two utility modules, respectively the validator (which has 100% coverage on all statistics), and the logging manager. The latter is not part of production code (as logging is disabled, and merely a debugging feature). In conclusion, the server is extremely well tested. In overall, it has high statement (96.06%), branch (80.15%), function (94.59%), and line coverage (96.06%).

[2] This low percentage is partially due to the large quantity of comments.
[3] The line amounts include comments and white-space.

5.4. INTEGRATION TESTING RESULTS

Integration testing takes place between the different system components, but also between sub-components in a system component. The integration tests take place in a developer-controlled environment. Each of the integration tests were executed multiple times (following the guidelines 5.2.1). In section 5.4.1 up to section 5.4.5, it is for each integration test explained what it entailed, and which information was derived from it.

5.4.1. SMARTPHONE UI

In this integration test, there is an exhaustive interaction with the smartphone application's functionality through the user interface (UI). This includes enabling and disabling long range communication capabilities.

The UI of the smartphone is not particularly a part of the system design, but is responsible for the correct input of the device location and information, not to mention (de)activation of the sensor node. It was important that the functionality behind the UI maintains state consistency. Different input styles, and button interaction sequences were performed. One of the most important bugs that was discovered was in the usage of Java's *Scanner.nextDouble()* function, that requires a Locale to parse doubles (English using a dot, and Dutch a comma).

5.4.2. SMARTPHONE DATA PERSISTENCE

In this integration test, the server is run and the sensor node is activated on the smartphone. Subsequently the logging information shown in the application is matched with the data in the database.

By performing this test, the implementation of the server's API on the Android smartphone was tested. This test was particularly helpful when the API changed, and thus the implementation on the sensor nodes required to be adjusted. These adjustments were then validated, giving insight into the correctness of the API, and its implementation on the Android smartphone.

5.4.3. COMPONENT DRY-RUN

In this integration test, each system component is run without any other components active. This tests their autonomous capabilities, and that each system component does not show unexpected behavior due to dependence on other system components.

For the system to perform its function, the system components need to collaborate. However, the system must be able to handle the failure of devices on which the system components are implemented. An example of this occurrence came forth when the Bluemix server was undergoing maintenance, and therefore unreachable for both hub and sensor nodes.

5.4.4. ARDUINO TO UNCONNECTED HUB

In this integration tests, the arduino is run without a connection to an hub. It tests that the Arduino does not show unexpected behavior when connected to a Hub, which is not connected to the server.

The hub functions merely as a channel between sensor node and server. If either end is interrupted, both sides of the connection must be able to handle the lack of connectivity. This test proved especially useful in testing the buffering behavior of Arduino sensor nodes.

5.4.5. ARDUINO DATA PERSISTENCE

In this integration test, the server and hub are run, and the Arduino is activated. Subsequently the logging information produced by the Arduino sensor node and the hub node is matched with the data in the database.

The Arduino interacts through a hub node with the server. All of the data received from the Arduino sensor node, complemented with the meta-data from the hub node, must be handled according to the API definition. When the API changed, this test proved particularly useful in discovering implementation errors on both the Arduino sensor node and the Java hub node.

5.5. SYSTEM TESTING RESULTS

System testing is performed at the end of the development cycle. It involves unleashing the system into the environment it was designed for. In the case of this demonstrator, the office of IBM CAS. This is the final test of the system, as it involves all system components running together in a distributed fashion. First, in section 5.5.1, an overview is given of the deployment. Second, the results of the preliminary deployment are discussed in section 5.5.2. Third, the results of the main deployment are discussed in section 5.5.3.

5.5.1. DEPLOYMENT OVERVIEW

Both deployments were performed at the Centre for Advanced Studies at IBM, Amsterdam. First, the environment of the deployment is described. Second, the strategy is elaborated upon.

ENVIRONMENT

The environment is an office space. An overview of the office space is depicted in figure 5.3. There are a total of 22 desks on which the system can be deployed. It is expected that only a portion of the employees are attending, or have the required hardware with them to accommodate the Arduino or Android sensor node implementation. The hardware that has been tested for specifically, is the standard Lenovo T430 laptop. The location of the Arduino sensor nodes is fixed, and these low-cost sensor nodes are supposed to be paired together with their complementing Android sensor nodes.

Figure 5.3: Floor plan of the low-cost sensing devices deployed at the second floor of IBM CAS in Amsterdam

STRATEGY

Each deployment was performed using the following strategy:

- **Arduino Upload:** The Arduino sketch is uploaded to each Arduino sensor node, with its device specific information (e.g. name and static location).
- **Location Notes:** On each measurement spot a note is put, displaying its coordinates. This manual approach is done due to the inability of the sensor nodes to dynamically sense their location, at this stage of the project (see also future work section 8.3.3)
- **Hub Nodes:** The hub devices are configured and paired with their respective Arduino device.
- **Android Sensor Nodes:** The participating Android devices install the application and configure their settings (e.g. name and static location).
- **Activation:** All hub and sensor nodes are switched on, the data collection begins.

5.5.2. PRELIMINARY DEPLOYMENT (JUNE 12)

On June 12, a first deployment was conducted in the controlled environment of IBM CAS. After going through the strategy, the experiment started in the afternoon. Three types of hub nodes were present: several Lenovo T430 laptops, an Asus N-series laptop, and a HP 8570w laptop. All hub nodes were responsible for the connection with a single Arduino Nano ATmega328 low-cost sensor node. Two smartphone sensor nodes were deployed as well, being a Samsung Galaxy S5 and a Google Nexus 5. The sensor nodes generated 11,030 measurements over the span of approximately two hours. The deployment went smoothly, especially considering it was the first deployment on 'foreign' hardware. The preliminary deployment statistics are depicted in table 5.4.

Nr.	Loc.[4]	Measurements	Device Type	Measurements per hour	Timespan[5]	Inconsistency cause
1	#18	3489	Arduino Nano ATmega328	1789	1:57	
2	#17	3303	Arduino Nano ATmega328	1852	1:47	
3	#12	2868	Arduino Nano ATmega328	1792	1:36	
4	#17	109	Android (Samsung Galaxy S5)	327	0:20	User behavior[6]
5	#18	1261	Android (Google Nexus 5)	713	1:46	

Table 5.4: Statistics Preliminary Deployment

5.5.3. MAIN DEPLOYMENT (JUNE 15)

On June 15, the main deployment was conducted at IBM CAS. The strategy actions were performed faster than the first deployment, as expected. Due to the absence of employees, only a part of the maximum potential in terms of amount of sensor nodes was achieved. Three types of hub nodes were present: several Lenovo T430 laptops, Asus N-series laptop, and HP 8570w laptop. A total of 11 Arduino sensor nodes, and 2 Android sensor nodes participated. Deployment on one other Android tablet device (Asus ME173X) was found to be unstable and is taken into account in future work, section 8.3.9. Over the course of roughly 4 hours, a total of 72,819 measurements were recorded by the participating 13 devices. The main deployment statistics are depicted in table 5.5.

[4] Location can be found in figure 5.3
[5] Time difference between the last and first recorded measurement.
[6] This was caused by it being only used very shortly, being switched on and off repeatedly, while simultaneously loosing WiFi signal. This was done by one of the team members as a test
[7] Location can be found in figure 5.3
[8] Time difference between the last and first recorded measurement.
[9] When the participant accidentally shut down the hub, the attached sensor node was unable to send its measurements to the server. When this was noticed by the development team, the hub was rebooted, but time had passed
[10] Repeatedly switched on and off. This was done by a team member to test the Android smartphone
[11] During lunch break, the app was switched off, but was not reactivated after lunch. The participant switched on/off at the end of the day, causing a very late last recorded measurement

Nr.	Loc.[7]	Measurements	Device Type	Measurements per hour	Timespan[8]	Inconsistency cause
1	#17	615	Arduino Nano ATmega328	1845	0:20	
2	#18	11898	Arduino Nano ATmega328	1802	6:36	
3	#3	8159	Arduino Nano ATmega328	1813	4:30	
4	#4	8678	Arduino Nano ATmega328	1771	4:54	
5	#5	8363	Arduino Nano ATmega328	1730	4:50	
6	#6	2703	Arduino Nano ATmega328	1763	1:32	
7	#12	5723	Arduino Nano ATmega328	1073	5:20	Signal loss
8	#11	2050	Arduino Nano ATmega328	572	3:35	User behavior[9]
9	#9	6725	Arduino Nano ATmega328	1,143	5:53	Signal loss
10	#7	10093	Arduino Nano ATmega328	1745	5:47	
11	#15	6274	Arduino Nano ATmega328	1094	5:44	Signal loss
12	#18	401	Android (Google Nexus 5)	414.83	0:58	User behavior[10]
13	#17	1137	Android (Samsung GT-I8190N)	144.84	7:51	User behavior [11]

Table 5.5: Statistics Main Deployment

5.5.4. DEPLOYMENT ANALYSIS

In both deployments, the Arduino sensor nodes had three sensors (temperature, humidity and light intensity) attached to them. All smartphone sensor nodes had a single sensor switched on, which measured the battery temperature. All sensors were set to produce measurements every 5 seconds. This sums up to a maximum potential of 2160 measurements per hour for Arduino sensor nodes, and 720 for Android sensor nodes.

In the *measurements per hour* column in both table 5.4 and 5.5, it can be seen that some inconsistencies occurred during deployment. These are explained by the phenomena stated below. For each of these phenomena, a (reference to an) action plan is also discussed.

- **Buffering Effect and Communication Overhead:** The Arduino sensor nodes have to buffer their measurements, and then sent them to the hub. The buffer has a very limited size (10) due to memory capacities. The hub only engages in a connection in a fixed interval. When the buffer of the sensor node is full, it will adopt a first-in-first-out strategy. Thus the measurements per hour for the Arduino sensor nodes are less than the maximum potential. This effect could be mitigated by increasing buffer size, which would require to use other (more expensive) hardware, resulting in a consideration between cost and reliability. This is described in future work section 8.3.4.

- **Signal loss:** The Arduino sensor has to communicate continuously with the hub to store its measurements on the server. It was observed that there were periods in which the hub's internal Bluetooth module and Arduino Bluetooth module were unable to connect with each other (which is handled by the third party module firmware). Additional tests for both the hardware and software of the hub nodes and Arduino sensor nodes is needed to analyze their reliability. This is addressed in future work section 8.2.1.

- **User behavior:** User behavior can influence the measuring. If the sensor node, or hub node, is shut down by the user, no measurements will be taken or channeled. This could be mitigated by connecting the sensor node to multiple hubs, or by warning the user that closing the hub program window will end his participation in the demonstrator (see section 8.4.2). Furthermore, a system overview interface will allow for better monitoring of the system, as is described in future work section 8.5.2.

5.5.5. DEPLOYMENT CONCLUSION

The two deployment tests yielded promising results, while also uncovering possible improvements in the system. These improvements range from enhancing the communication hardware to reduce signal loss, optimizing the communication overhead, increasing buffer size, and improving the usability of applications that

require user interaction, namely the smartphones and hubs. In conclusion, both deployments are considered successful, as they already produced valuable measurement data in this early demonstrator phase.

6

Final Product Evaluation

6.1. Introduction

Throughout the process of developing the product, we reviewed the requirements on a regular basis, when discussing our sprint plans. During the development process we also took into account the quality of our code, through the usage of several testing and analysis tools as will be discussed in this section. We thus respectively both validated [24] and verified [24] our product. By this, we ensured the quality of our product, of which an evaluation will be done in this section. First, in section 6.2, fulfillment of the requirements is evaluated. Second, in section 6.3, the code quality is assessed.

6.2. Requirements Evaluation

In the research phase, a large set of requirements was established (see section 2.4). The fulfillment of these requirements will be evaluated in this section according to the success criteria (2.4.6). This evaluation of fulfillment ensures that the client receives the product that was promised at the start of the project, as was expressed in the requirements. In this section, each of the categories of requirements is evaluated separately. For each of the requirements their fulfillment is checked, and the explanation is given. Because requirements are subject to change, some requirements have been dropped, after discussion with the client. A requirement can thus be fulfilled (✓), dropped (/) or unfulfilled (×). All of the six categories of requirements have been evaluated: the general (6.2.1), sensor network (6.2.2), low-cost sensing device (6.2.2), smartphone app (6.2.2), cloud application (6.2.3), and visualization requirements (6.2.4). Finally, an evaluation summary of the fulfillment of the requirements is given in section 6.2.5.

6.2.1. General Requirements

All general requirements, aimed at system level, have been fulfilled (see table 6.1). All of the encoded functions are accompanied by comments (G1) and the code of the system was developed according to the specified test plan (see section 5.2) (G2).

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
G1	All functions in the delivered code *must* be accompanied by a comment explaining its input, behavior, and output.	Modifiability, testability	FPT, DTM	✓
G2	The delivered code *must* be developed according to the test plan (to be) described in section 5.2	Maintainability, Reliability	FPT, DTM	✓

Table 6.1: Fulfillment of the General Requirements

6.2.2. SENSOR NETWORK REQUIREMENTS

Most of the sensor network requirements (see table 6.2) have been fulfilled. The sensor network is able to measure temperature, light intensity, humidity (S1, S2, S4) with an adjustable interval (S9), a fixed location (S7) and a system-wide synchronized timestamp (S10). Its code was made in a way, that it is easily adaptable to measure other phenomena (S6) and allows for the deployment of new sensor nodes within a few minutes (S11). A sensor node is also configured with less than ten device-specific values (S14). Furthermore, the sensor network is able to handle dynamic addition and removal of sensor nodes (S15) and delivers its sensing data to the cloud application (S12).

The three requirements that were not met, were (a) the requirement that the sensor network should be able to measure the level of sound (S3), (b) the ability of the sensor network to measure the amount of people in the proximity (S5) and (c) the requirement concerning the CIAA of the system (S13). Although the low cost sensor nodes were implemented with the possibility to measure sound intensity, the measurements from this sensor seemed to be of no value, due to its limited range. We therefore chose not to implement this sensor in the demonstrator and dropped the requirement in consultation with the client. Second, the requirement concerning measuring the amount of people in the proximity of the sensor nodes was not implemented as it was found to be quite a stand-alone feature, costing quite some effort to implement while being of not that high value. However, an approach towards implementing this feature is described in section 8.3.8. Third, the deployment of the demonstrator took place in a controlled and secure environment. Because of this, low priority was given by the client to fulfill the CIAA requirement, also due to time constraints. Moreover, the system was designed to support this feature in future work (see section 8.2.3).

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
S1	The sensor network *must* be able to measure temperature	Functional completeness	RJS, UR	✓
S2	The sensor network *should* be able to measure light intensity	Functional completeness	RJS	✓
S3	The sensor network *could* be able to measure the level of sound	Functional completeness	RJS	/
S4	The sensor network *should* be able to measure the level of humidity	Functional completeness	RJS	✓
S5	The sensor network *won't* be able to measure the amount of people in the proximity of sensor nodes	Functional completeness	UR	×
S6	The sensor network *should* be adaptable to measure other phenomena	Functional completeness, Adaptability	RJS	✓
S7	The sensor network *must* be able to measure the stated phenomena using measurements from sensor nodes with a fixed location (x, y, z)	Functional completeness	RJS, MO, UR	✓
S8	The intervals in which phenomena are measured *should* be adjustable	Functional correctness	RJS, UR	✓
S9	The sensor data *must*, when stored, be labeled with a system-wide synchronized timestamp	Functional correctness	RJS, UR	✓
S10	The sensor network *must* enable new sensor nodes to be deployed within ten minutes	Adaptability	RJS, MO, DTM	✓
S11	The sensor network *must* be able to deliver its sensing data to the cloud application	Functional completeness	RJS	✓
S12	A sensor node *could* communicate with the cloud application while maintaining the confidentiality, integrity, authenticity and accountability of the messages.	Functional completeness, Authenticity, Accountability, Confidentiality, Integrity	RJS, FC, SAP	×
S13	A sensor node *should* be configured with less than 10 device-specific values.	Installability, adaptability	RJS, MO	✓
S14	The sensor network *should* be able to handle dynamic addition, and removal of sensor nodes.	Adaptability	RJS	✓

Table 6.2: Fulfillment of the Sensor Network Requirements

LOW-COST SENSING DEVICE SPECIFIC

As can be seen in table 6.3, 3 out of 5 of the specific requirements for the low-cost sensing devices are met. The sensor nodes namely do not produce any audible sound at all (L1) and have an average price way below $80, as they have an average cost lower than $18, as was calculated in appendix E (L4, L5). The sensor network can also handle different positioning of the low-cost sensor node, but the nodes do not have a variable regarding their location to send (L3), though this is added to the future work (section 8.3.3). The dynamic removal and addition of sensors (L2) has been put to future work as well (see section 8.3.7).

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
L1	The low-cost sensor nodes *should* not make noise that disturbs employees.	Usability compliance	SAP, FC	✓
L2	The low-cost sensor nodes *could* enable the dynamic addition and removal of sensors.	Adaptability	RJS	×
L3	The sensor network *won't* be able to measure the stated phenomena using measurements from low-cost sensor nodes with a variable location (x, y, z)	Functional appropriateness	RJS, MO, UR	×
L4	The average price of a readily deployable low-cost sensor node, including supporting infrastructure, *must* be less than $100.	Expense	RJS, MO, FC	✓
L5	The average price of a readily deployable low-cost sensor node, including supporting infrastructure, *should* be less than $80.	Expense	RJS, MO, FC	✓

Table 6.3: Fulfillment of the Low-cost Sensor Node Specific Requirements

SMARTPHONE APP SPECIFIC

Almost all of the smartphone specific requirements have been met (see table 6.4). The application is installed on a voluntary basis (A1), and does not require any interactions except setup (A2). The app is designed using basic Gestalt design principles, and contains little UI elements. Futhermore, any information that can be derived from the device is already filled in for the convenience of the user (A3). The application matches the IBM colors, and is generally designed conform the IBM style (A4). The application code has been well tested (see section 5.3.1) and documented (see 6.3.2) (A5). Only A6, the ability to measure smartphones with a variable location has not been fulfilled. The data model and system however are designed to be able to handle variable locations, thus allowing this feature to be implemented in future work quite easily, as is elaborated upon in section 8.3.3.

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
A1	When interaction is required, it *should* occur on voluntary basis [7]	Usability compliance	FC, SAP, RJS	✓
A2	When interaction is required, it *should* occur in an engaging context. [7]	Usability compliance	FC, SAP, RJS	✓
A3	The app *should* be easy to use for a non-developer, minimizing the cognitive burden to the user [7].	Operability	SAP	✓
A4	The style of the app *could* match the IBM style.	User interface aesthetics	RJS	✓
A5	The mobile application *should* be well documented so that it can be implemented on different operating systems as well.	Reusability, Installability	RJS	✓
A6	The sensor network *should* be able to measure the stated phenomena using measurements from smartphone devices with a variable location (x, y, z)	Functional appropriateness	RJS, MO, UR	×

Table 6.4: Fulfillment of the Smartphone App Specific Requirements

6.2.3. CLOUD APPLICATION

As can be seen in the table below, all of the requirements specified for the cloud application have been fulfilled. In section 3.3 it is described that the data is stored in combination with the sensor type (C1) and it is described how sensor data is traceable to a specific sensor node (C9). In the same section, both textual and visual documentation for each entity and their relationships is provided (C2). Also, the current version of the system allows for monitoring states through server-side logging and their inputs which can be accessed from the database interface (C4), which is the IBM DB2 interface that is an IBM product (C3). As described in section 5.5, the cloud application can handle at least twenty nodes simultaneously (C5). Due to the usage of IBM Bluemix for running the server, it has become very scalable as well (C7). Based upon (C5) and (C7), we dare to be confident about the server being able to handle fifty devices simultaneously as well (C6). All of the endpoint code is completely documented (C8). The data model enforces measurement traceability (C9).

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
C1	The data *must* be stored in combination with sensor type	Functional completeness	UR, RJS	✓
C2	The data structure *must* have both textual and visual documentation for each entity, and the relations between entities	Modifiability	FPT	✓
C3	The style of the cloud application interface *could* match the IBM style.	User interface aesthetics	RJS, SAP	✓
C4	A basic cloud application interface that allows monitoring of the states of sensor nodes *must* exist	Analyzability, testability	DTM, RJS, MO	✓
C5	The cloud application *must* be able to handle at least ten sensor nodes simultaneously.	Functional completeness	RJS, UR	✓
C6	The cloud application *should* be able to handle 50 sensor nodes simultaneously.	Functional appropriateness	RJS, UR	✓
C7	The cloud application *should* be scalable, and thus scale with the addition of new resources.	Functional appropriateness	RJS	✓
C8	Every endpoint of the access API *must* be completely documented.	Modifiability	RJS, FPT, DTM	✓
C9	Sensor data *must* be traceable to a specific sensor node.	Accountability, Non-repudiation	RJS, DTM, UR	✓

Table 6.5: Fulfillment of the Cloud Application Requirements

6.2.4. VISUALIZATION

Due to re-prioritization in consultation with the client, no visualizations have been created (V1, V2, V3, V4). The focus in development was on creating a robust, reliable system. The ability to export data from the database is however present, which in turn can be used for offline visualization (e.g. graphs). In the future work chapter, section 8.6 is devoted to the implementation of this component.

Nr.	Requirement	Design goals	Stakeholder(s)	Fulfilled
V1	A simple statistics visualization *must* be developed	Functional completeness	RJS	/
V2	A graphical visualization *could* be developed	Functional completeness	RJS	/
V3	The style of the visualization *could* match the IBM style.	User interface aesthetics	RJS, SAP, FC	/
V4	Each visualization *must* (be able to) retrieve the visualization data from the server via the Internet.	Functional completeness	RJS, MO	/

Table 6.6: Fulfillment of the Visualization Requirements

6.2.5. EVALUATION SUMMARY

Finally, the project can be evaluated as a whole. By combining the evaluation for all system component categories, the entire system can be evaluated. The overall fulfillment by MoSCoW element is depicted in table 6.7. The overall fulfillment for each system component is depicted in table 6.8.

Category	**Total**	**Fulfilled**	**Dropped**	**Unfulfilled**	**Fulfillment[1] (%)**
Must have	16	14	2	0	100%
Should have	15	14	0	1	93%
Could have	7	2	3	2	50%
Won't have	2	0	0	2	0%
Total	40	30	5	5	86%

Table 6.7: Overview of fulfillment of the requirements by MoSCoW category

Category	**Total**	**Fulfilled**	**Dropped**	**Unfulfilled**	**Fulfillment[1] (%)**
General	2	2	0	0	100%
Sensor Network	14	11	1	2	92%
Low-cost	5	3	0	2	60%
Smartphone	6	5	0	1	83%
Cloud Application	9	9	0	0	100%
Visualization	4	0	4	0	/
Total	40	30	5	5	86%

Table 6.8: Overview of fulfillment of the requirements by system component

The evaluation by MoSCoW elements shows a healthy distribution. The expectations and success criteria described in the research phase in section 2.4.6 have been met. All MUST requirements have been met within the time frame of the project. Moreover, a large amount of SHOULD and COULD requirements are implemented, adding to its success. As was expected, adjustment of the requirements has taken place during development, mostly resulting in dropping some requirements due to re-prioritization. To give strength to this, 86% of all requirements - regardless of priority - have been met, resulting in a well-developed end product. In conclusion, the success criteria have been met, and were even exceeded.

[1] $Fulfillment = \frac{Fulfilled}{Total - Dropped} \times 100\%$

6.3. CODE QUALITY

Other than the requirement validation done in section 6.2, which is aimed at evaluating whether the product complies with the wishes of the client ("Are we the right thing?"), it is important to evaluate the quality of the code, which is commonly known as verification ("Are we building the it right?"). For this project we can even state that the latter may be of even higher importance, considering it concerns the development of a demonstrator which will be extended by future development teams. Future development teams of such a project should implement new features, instead of having to re-factor legacy code. It is important to note that (1) the requirements also pertain verification and (2) that code quality insurance through testing is described separately in chapter 5.

First, in section 6.3.1, the three tools used to measure code quality are introduced. Second, in section 6.3.2, the Sonarqube evaluation results are discussed. Third, in section 6.3.3, the feedback received from the SIG evaluations is discussed.

6.3.1. MEASURING CODE QUALITY

To verify the quality of the code, two methods are used: the static code analysis tool Sonarqube, and code submissions to the Software Improvement Group (SIG).

SONARQUBE

Sonarqube is an open source web-based application for managing code quality. It provides insight in the quality of the code regarding metrics such as complexity, duplications and code-style violations. Furthermore, it follows generally accepted coding conventions, making the code universally understandable. An example of the output produced by Sonarqube throughout one of the latest phases of the project is depicted in figure 6.1. In section 6.3.2 the Sonarqube results of the final code is provided separately for each of the system components, together with a description of and argumentation for the outcomes. An overview of these outcomes is provided in the latest paragraph of that section.

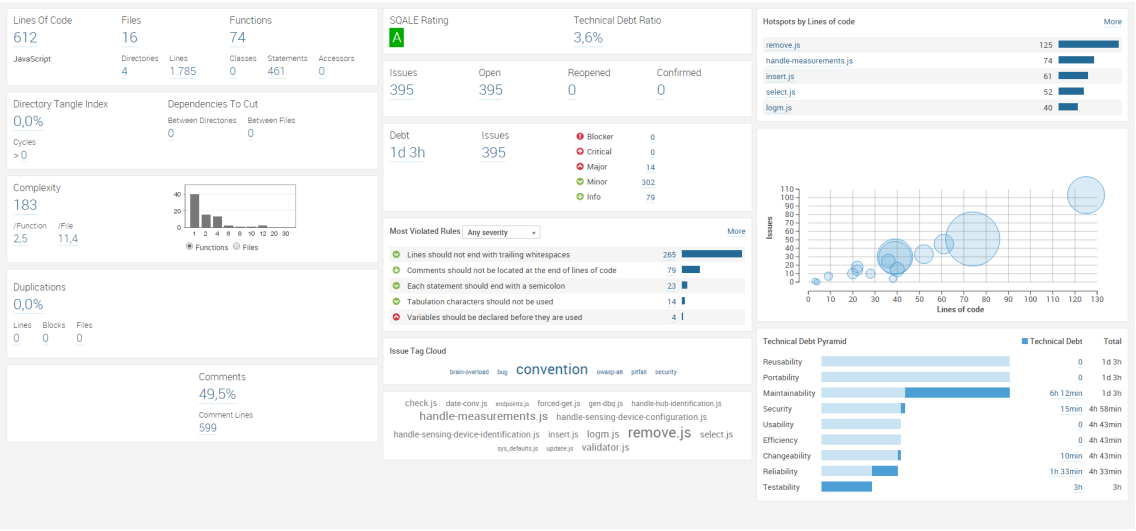

Figure 6.1: Sonarqube output for the Server NodeJS code in the second to last week of the project

SIG FEEDBACK

Two submissions were made to the Software Improvement Group (SIG). SIG is a management consultancy that focuses on the analysis of proprietary software, and acts often as an objective third party evaluator of projects. It originally stems from the Software Engineering department of the TU Delft. The submissions for code inspection were sent in the middle and at the end of the development process. SIG has provided us with feedback according to their 5-star classification system for software quality, based upon the ISO/IEC 25010 model. In section 6.3.3 our prospects, the actual feedback and our approach towards using this feedback is discussed for both submissions.

6.3.2. SONARQUBE

The Sonarqube tool was run for each of the four system component implementations. For each of these implementations the results are analyzed and discussed in this section.

ANDROID SENSOR NODE

The output of Sonarqube for the Android sensor node Java code is depicted in figure 6.2. The code scores an A at the SQALE rating, and has a mere 0.1% technical debt ratio. It is also well commented, consisting for 32.5% out of comments. From the metrics can be inferred that the hub node code is in excellent shape. Only a few minor 'issues' exist due to tabulation, which are deliberately unfixed as this tabulation is part of the code-style of the team (see appendix F).

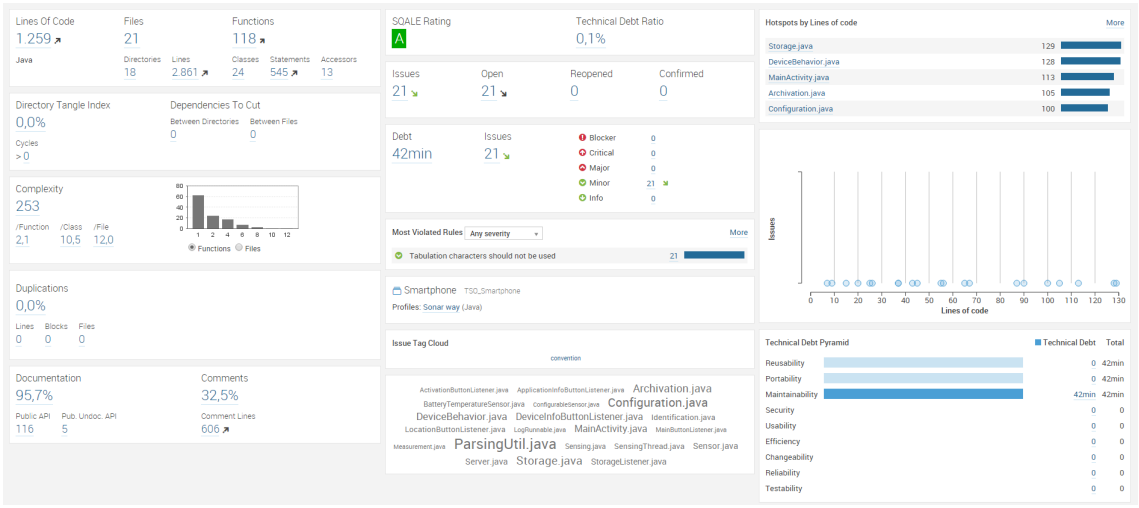

Figure 6.2: Sonarqube output for the final Android application code

ARDUINO SENSOR NODE

In figure 6.3, the limited output of Sonarqube for the Arduino code is depicted. Sonarqube does not provide as much information regarding C++/Processing code quality, as it does for Java. Though, we can see that the Arduino code generally seems to have sufficient commenting, being 37% of the code. Next to this, we can also find a complexity of 1.7 per function, which is quite low. We can therefore conclude that, based on the limited output, the Arduino code is in fine shape and of good quality.

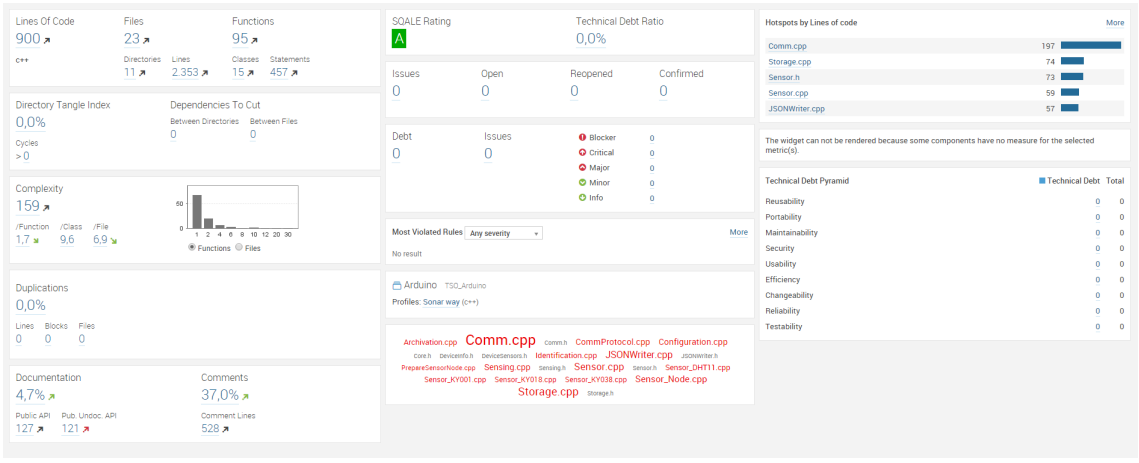

Figure 6.3: Sonarqube output for the final Arduino code

HUB NODE

The output of Sonarqube for the hub node Java code is depicted in figure 6.4. The hub node scores an A at the SQALE rating, and has a mere 0.2% technical debt ratio. It is well commented, consisting for 33% out of comments. From the metrics it can be inferred that the hub node code is in excellent shape, just like the code for the Android application.

Only two major issues reside, one concerning a constructor having too many parameters, and the other concerning complexity. The former is caused by the behavior constructor, which allows its caller to specify the behavior modules it depends on (e.g. identification, configuration). This many-parameter constructor is fully covered in unit tests, and therefore currently of low priority to adjust. The latter, concerning complexity, arose due to a logging function. Splitting it up into smaller functions would take significant re-factoring efforts, while yielding little improvement. The minor issues are only concerning tabulation, which is part of the code-style of the team (see appendix F).

In conclusion, the hub node code is in very good condition.

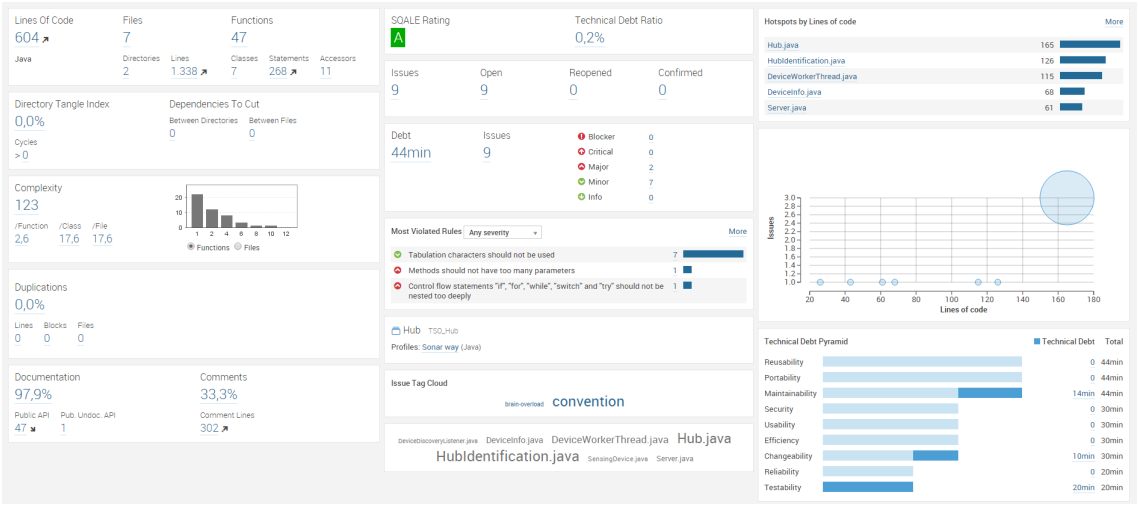

Figure 6.4: Sonarqube output for the final Hub code

SERVER

The output of Sonarqube for the server NodeJS code is depicted in figure 6.5. The server code scores an A at the SQALE rating due to its low technical debt ratio of 3,1%, is thoroughly commented (roughly 50% of the code consists of comments) and does not have any dependencies to cut, nor does it contain any duplications. From these metrics, we can generally state that the server code is of good quality. There are some issues detected by Sonarqube, in the three categories that are of lowest importance: major, minor and info.

There are two major issues. The first, the violation of the rule "console logging should not be used" is definitely acceptable for a logging manager file. Second, there is a concern regarding the complexity of certain handler functions. Lowering the complexity of the handler functions would indeed make the code more maintainable. The choice was made to not adjust this in the limited time span, as it takes considerable time, and the current code is easily understandable, well-documented, and well-tested.

Looking at the minor and info issues, we can easily conclude that the code-style check done by Sonarqube does not match our personal code-style. The code-style deviations are explained in appendix F. Examples include banning of tabulation characters, trailing white spaces, and trailing comments. The code-style is very personal, and adjusting it throughout the project would require a significant effort, while offering virtually no gain, but rather a loss of time and readability in our opinion.

In conclusion, it can be stated that the server code is of very good quality, as the metrics of the code are good. Less than 1% of the issues presented by Sonarqube can be seen as actual issues, of which a solution is provided in future work (section 8.5.1).

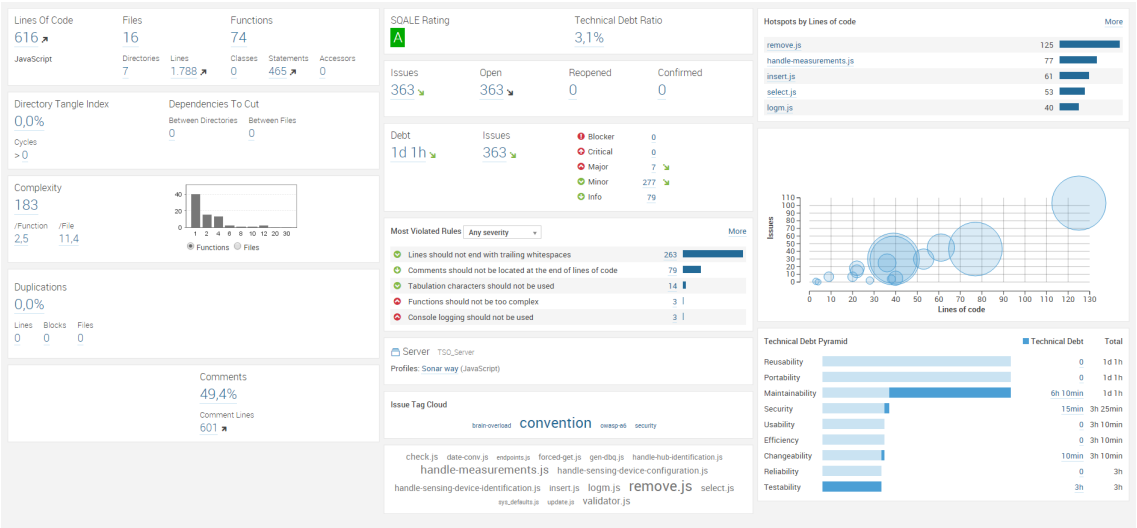

Figure 6.5: Sonarqube output for the final Server code

6.3.3. SIG EVALUATION

A total of two submissions to the Software Improvement Group (SIG) have bee done. The first was performed in the middle of the development, and the second towards the end. For each of the evaluations, the expectations were written down before the feedback was received. This can be contrasted by the reader with the actual feedback received. The actual feedback in the following sections has been summarized and translated. The raw feedback is documented in chapter G. The action plan (response) to process the feedback has been given for both evaluations as well.

FIRST EVALUATION

The first code submission was done on Tuesday May 26, the feedback was received Friday May 29. The code included all system components, excluding the Android application. The raw feedback can be viewed in appendix G.1.

- **Expectation:** The main code is of good quality, and proper coding conventions with regard to commenting have been followed. The test code has grown quite big, and due to shortage in time, we were unable to re-factor the test code before the submission. We therefor expect that in terms of unit size we will score sub-optimal.
- **Feedback:** The system scores four out of five on the scale of our maintainability model. In the C++ code there are many functions that have a boolean as final parameter, this should be reconsidered. Be wary of creating very long functions. Splitting up methods into smaller pieces will make the component easier to understand, test, and maintain. Sending the version of the project with and without libraries gives a good view of the project. Be sure to add which parts of the code are maintained by the project, and which by an external party. In general, the code scores above average. The presence of test code is promising, and we hope that this will grow as new functionality is added.
- **Response:** The C++ code will be re-factored to keep track internally of the state of the *JSONWriter* elements added. The server has been adjusted slightly to split up the endpoints into smaller pieces. An extra README file has been added to the libraries to distinguish between internally and externally maintained libraries. We will continue to thoroughly test our code.

SECOND EVALUATION

The second code submission was done on Tuesday June 9, the feedback was received on Wednesday June 17. The code included all system components. The raw feedback can be viewed in appendix G.2.

- **Expectation:** All system components have been tested thoroughly, and the code has been re-factored. Due to dependencies between components, there are still a few large constructors in the Hub system component, which might negatively affect out score on Unit Interfacing. The handlers of the server code are split up, although still larger than average functions, which probably will negatively influence the Unit Size. Furthermore, a lot of time was put into the new Android code. This code is additionally added to the second submission. The specific examples mentioned in the previous feedback have all been implemented. We expect the feedback to be slightly better than the first evaluation, considering the amount of unevaluated code that has been added since.
- **Feedback:** We see that in the second upload the code volume has grown, while the maintainability score has remained roughly the same. On the level sub-scores, we have a similar image as with the first upload: Unit Size and Unit Interfacing, which was marked as a point of improvement in the first upload, still score the lowest. On both aspects we see a small improvement, although this did not have a significant influence on the score. From these observations we can conclude that although the system is maintainable, the recommendations of the previous evaluation have been taken into account in a limited manner.
- **Response:** As expected, the Unit Size and Unit Interfacing has been improved slightly. Despite the addition of an entire Android application, the code has continued to be above average maintainable. In future work (see chapter 8), we have added the re-factoring of the server handlers, being the main point of issue for SIG, which we deliberately did not tackle due to the time needed to resolve the issue versus the quality improvements gained (which are quite low in our opinion). It is also important to note that the volume of tests has grown along with the addition of new functionality (including a thorough testing

suite for the Android application). Although this is not specifically mentioned in the second evaluation done by SIG, it is an improvement regarding the previous feedback gained.

7

PROCESS

7.1. INTRODUCTION

To give the reader a sense of the course of the system, this chapter is devoted to describing the development process. First, an overview of this process is described in section 7.2, providing the reader with an overview of the phases and sprints out of which the development process consisted. Second, the technical process is discussed in section 7.3, including a description of the tools that were used and technical challenges that were faced throughout the process. Finally, the actor processes are described in section 7.4, offering an insight in the interactions with the actors related to this project.

7.2. OVERVIEW

Throughout the development, a Git repository was used as a central point of project storage, enabling version control and synchronous labor. For each sprint, a new branch was created. This branch contained the progress of the project in that week, always resulting in a shippable product. The commits on the Git repository are an indication of how much implementation was done in a specific time period. The same indication is found for documentation in the number of pages produced. A total overview of the development progress, both implementation- and documentation-wise, is depicted in table 7.1. A graphical display of this table is shown in figure 7.1.

Sprint #	**Main Implementation Goal(s)**	**Main Documentation Goal(s)**	**# of commits**	**# of doc. pages**
1	Experiment with hardware	Research: Orientation	0	3
2	Experiment with hardware	Research: Analysis	1	10
3	Project Creation	Platform Dependencies	12	2
4	Hub, Arduino and Server Implementation	Data Model	37	11
5	Server Restructure, and Hub, Arduino and Server Testing	System Structure, Server API, Queries	46	15
6	Android, and Arduino Restructure	Sensor Node Structure	34	5
7	Android Testing, Minor Improvements, Report	Design and Implementation	14	30
8	Report, Minor Improvements	Implementation, Testing, Final Product Evaluation	4	30
9	Final Adjustments	Process, Future Work, Conclusion	3	22

Table 7.1: Development Progress Statistics

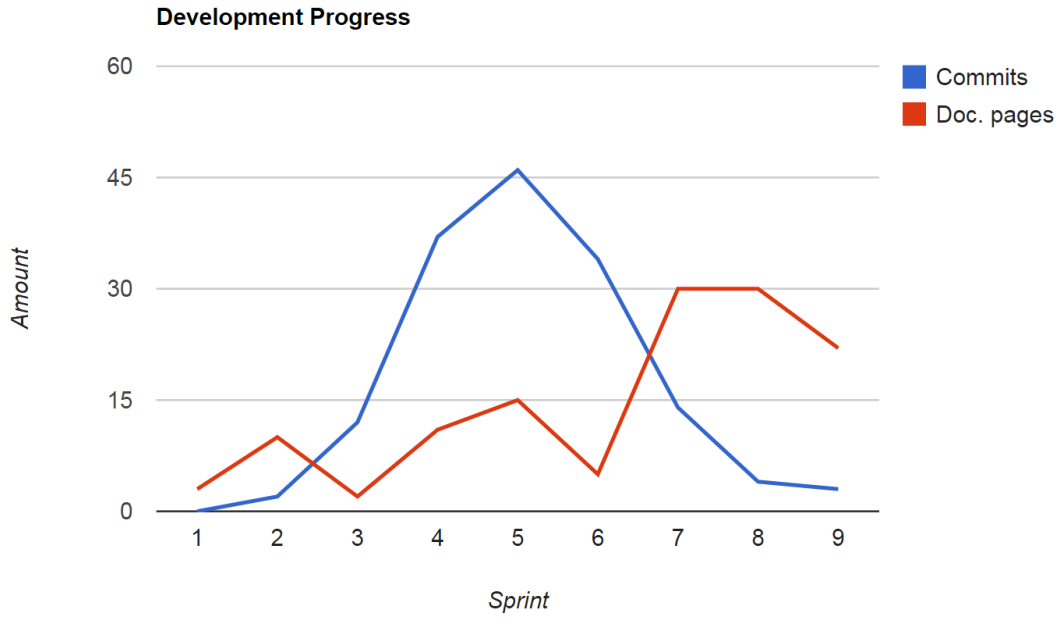

Figure 7.1: Development Progress Graph

The visualization in figure 7.1 especially highlights the diversity of phases that was undergone during the project. According to our plan, there was a three-phase-division, which we can also see coming forth in the data, being:

- **Orientation:** In the first three weeks, we mainly focussed on orientation, and produced little measurable value regarding lines of code and pages of reporting. However, in this time we collected a very thorough feeling of the problem, context of the problem and the wishes of the client regarding the solution. We also obtained invaluable personal skills, concerning development on completely unfamiliar hardware, e.g. Arduino, the Raspberry Pi, and Bluetooth modules. This led us to having everything we needed to start with the actual implementation of the project.
- **Implementation:** In the middle period of the project, we made a big shift from getting familiar with and investigating the problem and hard- and software to be used, to the actual implementation of the system. After the orientation, we knew how to properly write software for the enormous variety of platforms we had found to be needed. In this time, the main part of the implementation was done, after having carefully thought it out during the previous phase. Documentation was generated to accompany design choices, and to provide structure to the internal workings. Towards the end, we focused more on testing the system and streamlining out bugs.
- **Documentation:** After having largely implemented the system while generating temporary (non-final) documentation, the time arrived (due to time constraints) to properly document the system. By having worked meticulously in the previous phases, the report was already roughly written in our heads. During this phase, it was put to paper.

7.3. TECHNICAL PROCESS

In this section the technical process is described. First, in section 7.3.1, the development tools are evaluated, this includes both coding and documentation tools. Second, in section 7.3.2, the technical challenges we faced, are explained.

7.3.1. DEVELOPMENT TOOLS

A vast combination of tools was used during development. We were already familiar with the usage of the Eclipse IDE, JUnit, EclEmma, the Android SDK, EMMA, Notepad++, SourceTree, Trello, Google Drive, and ShareLatex. These tools performed well, as we expected based upon our previous experiences.

Regarding the tools we were unfamiliar with, we were especially impressed with IBM Bluemix, Mocha and Istanbul, and Sonarqube. We were surprised with the ease of application deployment through IBM Bluemix, and the ability to dynamically add services such as a database. The combination of Mocha, as unit testing framework, and Istanbul, as coverage tool, turned out to provide valuable debugging information and greatly improved test quality regarding the server code. The wide support for programming languages of Sonarqube allowed static code inspection of all code using a single tool. The suggestions Sonarqube made were language-specific, and improved code quality significantly.

Tools that were found to be less useful were IBM MobileFirst, a mobile platform-independent development platform, and Blanket.js, a coverage tool for JavaScript. The former required a large amount of packages to work, and seemed to be in a prototypical state. The latter turned out to be incompatible with the server's NodeJS code, despite the best efforts of the team.

7.3.2. TECHNICAL CHALLENGES

UNKNOWN HARDWARE

The team was previously inexperienced with the usage of a large part of the hardware that was found to be needed for this project. It was required to quickly get familiar with it, to evaluate its usefulness for the project, and to fit it into the system design. Reliability was one of the most challenging factors, especially when testing out communication modules such as the HC-05 and the ESP8266. The software tools that we used for the platforms, had specific qualities and conventions as well, which all needed to be understood in order for the development team to be able to properly write and debug code.

HARDWARE DELIVERY

The delivery time of hardware turned out to span a significant amount of time, relative to the duration of the project. Although there was hardware already present on-site, such as the Arduino Nano chips, a lot of other hardware components were not (e.g. temperature and Bluetooth sensors). This hardware also had to be tested, before a large quantity was ordered. This put heavy time constraints on when development and deployment could take place.

7.4. ACTOR PROCESSES

In this section we discuss the process of the actors involved in this project. We start out with discussing the actor processes within the team and with the client. Next, the process involving other IBM employees and interns is described, followed by a description of the process with the supervisor and coordinators from TU Delft. For some of the actors involved in this project, a description of their roles is given in section 2.2.4.

7.4.1. THE TEAM

The development team of this project consisted of just two members. This enabled us to cooperate thoroughly and communicate on a nearby 24/7 basis throughout the project, as we were working next to each other for nearly 90% of the time. It allowed us to instantly ask for each others opinion and to constantly be aware of each others' progress. Luckily, we did often have different opinions on certain design or implementation choices that had to be made, enabling a good discussion to be held. These discussions always ended in a consensus, which resulted in the best informed choice to be taken.

The software development method that was adopted by the development team concerned an adapted version of Scrum, as is described in section 2.6.1. This agile development methodology has proven to be a good

approach for creating a shippable product over a short period of time. The iterative structure of programming and regular meetings with the stakeholders involved in the project, enabled for a highly accessible development process at any point in time.

7.4.2. THE CLIENT

Throughout the project there has been a strong communication between the team and the client, in which we specify the client as IBM, impersonated by Robert-Jan Sips (our supervisor from IBM). We regularly held meetings, up to three times a week, to discuss the current state of the project and the next steps to take. These frequent meetings enabled us to both validate and verify the product with the client, making it continuously in alignment with the client's wishes. It namely enabled us to, at any point in time, spend our time on developing the features that were most valued by the client. Next to this, it allowed us to involve the client in making decisions in designing the system and implementing it according to his wishes.

7.4.3. IBM EMPLOYEES AND INTERNS

At IBM, we also held meetings with other IBM employees and interns. We held several meetings with Uditha Ravindra (who will use the output of the demonstrator), discussing the output of the system, so that it would contain the data she needed. We also held several discussions with Manfred (the technical supervisor of the project), regarding the hardware of the system. Throughout the project, we also interacted with other IBM employees and interns, sharing knowledge. This was valuable because they either work on related projects, or work with related tools. This enabled us to get quickly started with the project in the IBM environment.

7.4.4. THE SUPERVISOR

At least once a week, we also discussed the current state and process of the project with our supervisor from TU Delft. Sometimes this was done through emailing, other times through an actual (physical) meeting. In these discussions, our supervisor provided us with feedback about our approach for developing the system and documenting the outcomes. This guidance allowed for the confidence of adopting the best approach and taking into account all matters at stake.

7.4.5. COORDINATORS

Other than a few questions before and during the project regarding the regulations of the bachelor end project, we held one Skype meeting with the coordinators of the bachelor end project. From this meeting, we took away that we should not underestimate the effort needed for the report, testing and deployment. The coordinators will also be involved in grading the project.

8

Future Work

8.1. Introduction

At the beginning of the project, we set course for creating a demonstrator; a strong basis for a potentially commercial product that is to be developed for the client. In other words, we already knew that there would be future work to be done, which could not be taken into account during this project, due to its limited scope of ten weeks. In this section we will elaborate upon the future work that can be done in order to further improve the demonstrator to become an actual commercial product. This is expressed in (1) improvements to the existing code and (2) separate extensions to several components of the system. First, in section 8.2, the future work involving multiple components of the system, is explained. Second, in section 8.3, future work on the sensor node is discussed, for both the low-cost sensor node and the smartphone app. Third, in section 8.4 future adaptations and extensions to the hub node are elaborated upon. Fourth, future work on the server is discussed in section 8.5. Finally, the future addition of a visualization component is discussed in section 8.6.

For each of the items in each of the five future work categories, it is specified (a) what functionality could be added and (b) how this could be done with respect to the current implementation. This will enable future development teams to quickly enhance the product with these features.

8.2. General

This section is devoted to the general future work. This future work spans two or more of the system components. For each functionality is specified what it entails, and how it could be implemented.

8.2.1. Communication Reliability Evaluation

A choice was made to use low-cost Bluetooth modules for the Arduino sensor nodes. This enabled them to communicate with other Bluetooth devices that would take on the role of a hub. A protocol was devised to communicate between both nodes. This protocol, its implementation, and the reliability of the hardware should be tested and evaluated more thoroughly. The designed framework described in section 3, abstracts from the usage of Bluetooth communication, and allows the usage of other communication standards such as Zigbee or NFC. These other communication techniques might prove to enable for a more reliable communication. Environmental factors such as distance between sensor node and hub, and obstacles between, might affect the reliability as well. These factors should be studied, and taken into account more thoroughly, before the system can become commercially viable.

8.2.2. Efficient Communication Format

In the demonstrator, the JavaScript Object Notation (JSON) is used for messages sent between different system components. Although it allows for a clear visual understanding (i.e. the JSON notation is easy to read), this notation is quite inefficient compared to other formats. A more efficient (self defined) format could be used in the final system, e.g. making use of run-length encoding and integrity hashes.

8.2.3. CONFIDENTIALITY, INTEGRITY, AVAILABILITY AND AUTHENTICITY OF DATA

The current communication format does not contain any security measures. As this version of the system was purely a demonstrator, the system resided in a secure environment. When aiming at deploying the system outside a secure environment, additional layers must be added over the communication format, such as encryption and integrity checks. Extra communication protocols should also be introduced to enable protective measures such as key distribution. This improvement must be done system-wide, and will require vast changes to the sensor node (in the communication module) and server implementation (across all existing endpoints, but also the addition of new endpoints).

8.3. SENSOR NODE

8.3.1. HEARTBEAT

Although measurements do already function as a heartbeat in most cases, there are edge cases in which the sensor node might be active, but does not produce any measurements. This can for example occur when all sensors have been disabled, or malfunction. The addition of a heartbeat into the protocol will allow the server to more easily detect which sensor nodes are active at which moment in time. Implementing such a heartbeat would ask for the addition of an extra endpoint (e.g. */heartbeat*) to the server, and the addition of a communication protocol (or extension of the existing communication protocol) that allows for an efficient communication with this endpoint. The protocol should entail a short, simple message being sent regularly (in a fixed interval) by the sensor node to the server's heartbeat endpoint.

8.3.2. ADDITIONAL PHENOMENA

Currently, the low-cost sensor nodes and smartphone application respectively measure the temperature, light-intensity and humidity, and the temperature of the battery. As said in the COULD requirement S3 in section 2.4.3, the sensor nodes could also measure the level of sound. Specifically for the smartphone app, the light-intensity and humidity could also be taken into account. This would allow for setting first steps into the direction of being able to doing the measurement with (calibrated) smartphone sensor nodes only, as stated in section 2.2.3.

Implementing sensing functionality for the additional phenomena can be done very easily. Regarding the low-cost sensor node, this would only ask for adding a new physical sensor to the breadboard, connecting this to the Arduino Nano, adding the sensor handler to the Arduino Nano code (e.g. *SENSOR_XYZ.cpp*, which inherits from *Sensor*) and adding where the sensors have been attached in *DeviceSensors.h*. Concerning the smartphone app, this would only ask for adding a new Sensor inheriting class to the *com.ibm.tso.app.sensors* package and adapting the constructor of the *DeviceBehavior* class in the *com.ibn.tso.app.behavior* package to add an additional sensor of the newly specified class. For both cases we predict it would not ask more than one hour to implement the sensing of a new phenomenon (excluding time needed for writing tests as this may be strongly dependent of the type of the sensor).

8.3.3. VARIABLE INDOOR LOCATION

The location of the sensor nodes is currently put in manually. Variable indoor localization can potentially be achieved by deploying Bluetooth beacons. These Bluetooth beacons will broadcast their location and environment, through a Bluetooth signal. The sensor node can combine the broadcasted information with the signal strengths to determine both its position (via trilateration) and environment (which is included in the information). In order to implement this feature, a developer would have to adjust the behavior of the sensor node. The sensor node will require a new class (e.g. *LocationFinder*) that uses the Bluetooth module to pick up and process the broadcasted information. This class will need to continuously check for changes, and supply the most recent location to the *Sensing* class.

8.3.4. LOW-COST SENSOR NODE CUSTOM HARDWARE

The low-cost sensor nodes in the current demonstrator were built out of general purpose hardware, such as the Arduino Nano chip. This general purpose hardware contains many components that are not needed (as indicated by its name), but do increase cost. Furthermore, the buffers size on the Arduino Nano chip is very limited. An electrical engineer could design a far more efficient electronic design, which implements exactly all the hardware required. The production cost of this custom design would be significantly less, and its performance (e.g. in terms of energy consumption, communicative capability and buffer size) would be significantly better. However, this would ask for a quite large investment regarding the design of such a

system. Moreover, this would also negatively affect the extendability of the system. Whereas the Arduino Nano Chips on breadboards enable for easy extension of the node to include new sensors, a dedicated board would not allow for such an extension. Porting the existing software to this new hardware is expected to be of a lesser concern, due to the high portability of the implementation.

8.3.5. SMARTPHONE PLATFORM INDEPENDENCE

The deployment of the system on many platforms is a strongly desired feature, as it would immediately allow for the adoption of far more smartphone sensor nodes in the sensing network. The current implementation of the Android application could be ported to a platform-independent environment, such as IBM MobileFirst, which supports many platforms (e.g. Android, iOS, Blackberry, Windows, and as a web application for others). This hybrid application would be implemented both platform-independent (e.g. connecting to the server) and platform-specific (e.g. reading the device sensor). The client has indicated that the usage of IBM MobileFirst has strong preference over the usage of other hybrid environments. A separation between destined platform-specific and platform-independent implementation must be made in the existing Android code, and the code destined to be platform-independent should be implemented in the general language of the hybrid framework (e.g. JavaScript).

8.3.6. DEFECT DETECTION

In the deployment tests, no defects occurred. This is however attributed to the controlled, safe environment in which the deployment tests were conducted. An implementation can be made so that sensor nodes can detect a failing module. With digital sensors, this could be detected by unresponsiveness. With analog sensors this might be more difficult, as a specific output range check would need to be conducted for each type of sensors and sensing device configuration. The data model and the server both support the (dynamic) configuration of sensors (through the *SensorActivity* and *SensingDeviceConfiguration* entities), but the sensor nodes do not yet utilize this support.

8.3.7. DYNAMIC ADDITION AND REMOVAL OF SENSORS

The support for dynamic addition and removal of sensor would allow the sensor nodes to become more autonomous. The data model already supports this, through requesting a new configuration from the server. This dynamic addition and removal can be implemented by checking on a fixed interval whether new sensors have been added to ports, or existing sensor have been removed. Communication must take place to determine the sensor type. This will prove to be more difficult with (analog) sensors that do not provide additional information about themselves. A new class would need to be made responsible for this regular checking (e.g. *SensorDetector*), but on forehand, investigation has to be done into the possibility to fulfill especially the dynamic addition of sensors.

8.3.8. PROXIMITY DETECTION

The detection of nearby entities such as humans would allow the sensor node to adjust its measurements. This proximity detection could be done through the addition of ultrasonic sensors, or through detecting patterns in the measurements that those entities affect. Examples include the sudden increase in temperature due to the presence of body heat, or the shadow entities cast on a light sensor. This feature is expected to be quite difficult to implement. Regarding the second option for implementation, it would definitely benefit from machine learning.

8.3.9. BROADER PLATFORM TESTING

The application was developed with support for Android devices running the Android OS of version 2.2 and above. Throughout this project, we only tested devices running Android 2.2, 4.1, 5.0 and 5.1. Additional testing for devices running other versions of Android is recommended, as such devices can also become part of the system. Next to this, testing could be extended to include other device types, like tablets, as well. This would allow the future scope of the sensor nodes to be extended to other IoT devices as well, which is a desired feature as discussed in 2.2.1.

8.4. HUB NODE

8.4.1. PARALLEL BLUETOOTH CONNECTION HANDLING

The hub has been implemented in a multithreaded fashion, allowing simultaneous handling of sensor nodes connections. On the contrary, the Bluetooth adapter only allows for a single Bluetooth connection to be open at one point in time. Problems such as starvation could occur, although they are highly unlikely to appear. Addition of a Bluetooth adapter that can handle multiple simultaneous connections would significantly increase the response speed of hub nodes, and also slightly increase reliability. For this feature to be implemented, the hub node code would require little adjustment, as it is already implemented multithreaded. However, support of the BlueCove library for the associated Bluetooth stack for opening multiple connections should be checked.

8.4.2. HUB USER INTERFACE

The hub implementation could benefit from an interface that displays how many sensor nodes are connected to it, and what their latest activity was. Furthermore, this interface can prevent its host from unintentionally shutting down the hub node. This interface could be implemented using the Javax Swing library, which is included by default in Java distributions.

8.5. SERVER

8.5.1. REFACTOR HANDLERS

In order to lower the complexity of the NodeJS handler files, they should be refactored. The four current handlers - especially */measurements* - have a high cyclomatic complexity due to the checking of received identifiers. Conversion of the handlers large functions into several smaller functions will increase maintainability of the code.

8.5.2. MANAGEMENT AND MONITORING SYSTEM

It is currently only possible to manage and monitor the system through the database interface. Addition of (for example) a web interface, would enable administrators to more easily manager and configure the sensor and hub nodes. This was not done within the scope of our project, as more importance was given to the actual sensing system and the ability to configure this system, rather than creating a proper interface (which, if done well, would require significant design investment). A HTML5/JavaScript website could do this, making asynchronous requests to retrieve real-time data from dedicated server endpoints.

8.5.3. TRANSACTIONALIZE ENDPOINTS

The server endpoints currently share a database client connection, which makes it unable to fully transactionalize endpoints. With the cost of losing speed, each endpoint call could create its own database client connection. This could also reach the ceiling of allowed database connections (determined by the database). Making every endpoint handling a transaction, will increase the consistency of the data stored persistently. An example would be when the server loses power, and already inserted a few of the measurements, but not all of them. If this were a transaction, the inserted measurements would be rolled back instead of wrongfully persisted in the database. This can be implemented by starting a transaction at the beginning of the handler, and by rolling back at a failure, or committing if it was a success.

8.5.4. TIME SYNCHRONIZATION

The addition of a central server which hands out the current time. This will especially come into play when the server is deployed on many servers, which all require a synchronized sense of time. The times synchronization server could be completely dedicated, only having an endpoint */time*. The other servers will then request a synchronous timestamp from the dedicated time server when needed.

8.5.5. DATABASE INDEPENDENCE

The server implementation currently only supports DB2 relational databases. However, due to the existence of a separate database query module, it is possible to support other relational databases as well (e.g. PostGreSQL and MySQL). Two adjustments need to be made: (1) the database query module must implement a storage class that stores the SQL queries for the chosen database type, (2) the database client must become

an abstraction, of which an individual implementation must be made for each relational database type. This abstracted database client should have a *connect()* and a *querySync()* function.

8.5.6. DATABASE IMPROVEMENTS

In section 4.5, some possible improvements for the data(base) model have been proposed for several tables:

- The addition of a descriptive column in the *SensingDevice table* for the operating system, which is being run on the device.
- Determination of what makes an *Environment* unique, instead of solely relying on the generated identifier. This is difficult, because it is not yet known where the environments will be. Identifying information such as "Company" might not be applicable in all situations.
- Linking the *DeviceType* with external databases containing information about hardware instead of relying on devices to provide accurate information.
- Extra descriptive information about a *SensorType* could be added, such as precision, or scale.

It is up to future development teams to determine the value of implementing these improvements, as they are not system-critical.

8.6. VISUALIZATION

Due to a shift in priority, the visualization of the system has moved to future work. The system would benefit from two types of visualization: passive and active visualization.

8.6.1. PASSIVE VISUALIZATION

Passive visualization is performed on a full data set over a span of time, instead of only using the latest values. The passive visualization is more focused on aggregate statistics, and supporting claims concerning a larger span of time. One of such claims could be the existence of a correlation between sensor values (e.g. between light intensity and temperature).

8.6.2. LIVE VISUALIZATION

Live visualization receives continuous real-time data, and focuses on the most recent data it receives. It is a depiction of the output of the system in a very short period of time. The purpose of this live visualization is to keep track of the system, and to allow system administrators to detect errors (e.g. if a sensor falls out, the live visualization should highlight this). It would furthermore be of value as an interesting visualization when displayed on a screen in the same environment, attracting participants for the project while being in a prototypical state and gathering support while being a commercial product.

9

CONCLUSION

In the short span of ten weeks, an ambitious project has been completed. Through iterative development, and frequent interaction with the client, IBM, we have implemented a demonstrator system to measure several phenomena in the enterprise environment. The project was both a software engineering and research challenge, making it unique in the landscape of bachelor end projects. Moreover, a part of the system is pending to be patented. The demonstrator acts as a valuable asset to the research goals of the client, having met the success criteria by fulfilling all mandatory requirements and many optional requirements.

The value of the demonstrator was supported by the three phases undergone. First, a thorough analysis of the client's wishes was made, ensuring verification of the product. Second, we designed and implemented the system while maintaining running documentation, ensuring the validation of the product. Third, we carefully documented the system, ensuring that the client would be able to understand and build upon the system after delivery.

One of the remarkable features of the system is its cross-platform design, due to the uniformity of the interface across platforms. The level of abstraction allows any system component (sensor node, hub node, and server) to be implemented straightforwardly on any platform, by using the software design presented in this report. The system design can thus act as a framework for measurement systems, applicable outside the scope of this specific context.

The client imposed strict wishes regarding the data model. Initially envisioned as 'one big table', a relational data model has been designed. The data model intelligently couples the measuring data to a traceable state of the system. The focus on traceability in the implementation allows for time-sensitive data analysis afterwards.

Special effort has gone into ensuring the maintainability of the system. By writing test code, measuring coverage, performing static code evaluation, and third party evaluation, we have created high quality source code. This matches the goal of a demonstrator: to be easily continued upon.

To demonstrate the ability of the system, it has been deployed in two instances at IBM CAS. Both deployments were considered successful, as they already produced valuable measurement data in this early phase. Additional testing and improvements are recommended to increase the reliability of the system. By combining cross-platform design, traceable data modeling, and following best coding practices, a valuable demonstrator for data collection in the enterprise environment has been developed. The team is excited for the demonstrator to potentially be transformed into a commercial product by future development teams.

A

Initial Project Description as Proposed by IBM in March 2015

A.1. Background

Ubiquitous Computing and the Internet of Things are promising to have a disruptive effect on various domains [6]. One of such domains is the Enterprise Environment, where it gets increasingly important for companies to sense and personalize the (physical) working environment [7]. Effective personalization and sensing of an office environment requires highly dense data, on both a spatial and a temporal dimension. E.g., one would need to know how light intensity and temperature change throughout the floor of a building to enable either an appropriate seating advice, or an appropriate modification of the temperature. Current Building Management Systems do not allow for this detailed sensing, and are very expensive to change. Promising developments have been demonstrated in the fields of Agriculture, where low-cost sensing is allowing for highly dense sensing in remote locations [9]. Moreover, Overeem et al. [10] demonstrated a system in which data from embedded mobile phone sensors was successfully used to estimate ambient temperature.

A.2. Project

The focus of this Bachelor Project is to create a demonstrator system in which these sensing technologies will be employed in an enterprise environment. The system contains of 3 core components:

- Mobile Phone App: sensing ambient parameters, such as temperature, humidity, light intensity, noise through the embedded phone sensors.
- Low-cost sensing devices: either Arduino or RaspPi based, measuring air quality, humidity, light intensity, etc.
- Visualisation: a visualization component visualizing the state of a department floor as sensed by the app and devices.

The Demonstrator will run within the IBM building in Amsterdam, at the Centre For Advanced Studies. IBM personnel at the floor will be running the developed sensing app on their smartphone.

A.3. Research Questions (extra)

Example research questions are:

- Is it possible to correlate the highly coarse sensing through smartphones and/or low-cost devices to the measurements as done by the existing Building Management System?
- What is the optimal distribution between “fixed” low-cost devices and “mobile” sensors to guarantee output and accuracy while minimizing the amount of fixed sensors needed?

- What would be an optimal setup of such a sensing system: Cloud-Centric or Device Centric (the first meaning a centralized cloud instance calculating derivatives from the sensed parameters, the latter meaning local p2p networks of mobile and fixed devices, jointly deciding on what the most accurate measurement in a specific "zone" would be)?
- Which other parameters could be measured or derived from embedded sensors in smartphones.

A.4. DELIVERABLES

- Architecture document (containing requirement analysis + basic design decisions)
- Demonstrator code + Documentation Company Supervision & embedding

The project will be part of the "Inclusive Enterprise" research line within IBM. Supervision and support will come from the CAS Development team (2 architects, 5 BSc students), and the Innovation Center (5 developers). Materials such as IBM MobileFirst (development platform) and Raspberry Pi / Arduino devices will be readily available at the department. Example code on how to connect sensors to those can be found through IBM Developerworks, see e.g., http://http://www.ibm.com/developerworks/cloud/library/cl-poseidon1-app/

B

Quality Model of Design Goals

B.1. Functional Suitability

The ability of the system to satisfy stated or implied needs (from stakeholders).

Functional Completeness
The ability of the system to correctly perform the functionalities it was designed to perform.

Functional Correctness
The ability of the system to provide (data) services with the needed degree of precision.

Functional Appropriateness
The ability of the system to perform functionality appropriate to its specified tasks and objectives.

B.2. Maintainability

The ability of the system to allow maintenance and modification.

Modifiability
The ability for developers to modify the system, concerning factors such as code quality, decoupling, and cohesion.

Testability
The ability of developers and administrators to test the (new) functionality of the system.

Analyzability
The ability of administrators to monitor and analyze the performance of the system.

Reusability
The ability for developers to reuse (a component of) the system in building other systems.

Modularity
The ability of the system to be split up into discrete components.

B.3. Portability

The ability of the system to be ported to other contexts, or to handle a changing context.

Installability
The ability of the system to be deployed in a different environment.

Adaptability
The ability of the system to adapt to changing infrastructure, such as in the case with many IoT systems.

B.4. RELIABILITY

The ability of the system to perform functions under specified conditions within a specified time window.

Maturity
The ability of the system to perform its tasks under normal conditions.

Fault Tolerance
The ability to handle the occurrence of faults, e.g. failing devices or measurement errors.

Recoverability
The ability of the system, in the event of a failure, to maintain data consistency, and to return to a stable state.

Availability
The ability of the system to maintain the availability of services and data.

B.5. EFFICIENCY

The ability of the system to efficiently use what it has at it (potentially) has to its disposal.

Resource Utilization
The ability to efficiently use the hardware, and to account for the constraints that this hardware imposes such as durability and energy consumption.

Time Utilization
The ability of the system to perform the tasks with response and processing times and throughput rates within the imposed time constraints.

Capacity
The ability of the system to utilize a product or a system parameter to a maximum in order to meet requirements.

Expense
The ability of the system to be cost effective.

B.6. COMPATIBILITY

The ability of the system to exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware and/or software environment.

Co-existence
The ability of the system to function in a common environment with other products, without having a detrimental impact on any other product.

Interoperability
The ability of the system to let two or more components exchange information and make use of this exchanged information.

B.7. USABILITY

The ability of the system to be used by the specified set of users.

Operability
The ability of the user to operate the system.

Usability Compliance
The ability of the system to follow logical standards and conventions related to usability, influencing the willfulness of the user to use the system.

User Interface Aesthetics
The ability of the system to offer the user interactions in a pleasing and satisfying manner.

B.8. SECURITY

The ability of the system to actively or passively defend itself from dangers and threats.

Confidentiality
The ability of the system components to securely communicate with each other.

Integrity
The ability of the system to verify the integrity of data, that messages are indeed valid and untampered with.

Authenticity
The ability of the system to verify that the claimed origin is indeed responsible for the tasks done and data sent.

Accountability
The ability of the system to track the origin of data and performed tasks.

Non-repudiation
The ability of the system to enable the proving of actions or events to have taken place.

C

PostGreSQL Queries

C.1. Scheme Creation

```
CREATE SCHEMA TSO;
```

C.2. SensorType Table Creation

```
CREATE TABLE TSO.SensorType (
    SensorTypeID SERIAL NOT NULL,
    Model VARCHAR(127) NOT NULL,
    Quantity VARCHAR(127) NOT NULL,
    Unit VARCHAR(127) NOT NULL,
    DefActive BOOL NOT NULL,
    DefInterval INT NOT NULL,
    PRIMARY KEY (SensorTypeID),
    UNIQUE (Model, Quantity)
);
```

C.3. DeviceType Table Creation

```
CREATE TABLE TSO.DeviceType (
    DeviceTypeID SERIAL NOT NULL,
    Brand VARCHAR(127) NOT NULL,
    Model VARCHAR(127) NOT NULL,
    MarketingName VARCHAR(127) NOT NULL,
    Category VARCHAR(127) NOT NULL,
    PRIMARY KEY (DeviceTypeID),
    UNIQUE (Brand, Model)
);
```

C.4. ENVIRONMENT TABLE CREATION

```
CREATE TABLE TSO.Environment (
    EnvID SERIAL NOT NULL,
    Timezone VARCHAR(127) NOT NULL,
    Name VARCHAR(255) NOT NULL,
    Latitude DOUBLE PRECISION NOT NULL,
    Longitude DOUBLE PRECISION NOT NULL,
    Elevation DOUBLE PRECISION NOT NULL,
    PRIMARY KEY (EnvID)
);
```

C.5. SENSINGDEVICE TABLE CREATION

```
CREATE TABLE TSO.SensingDevice (
    DeviceID SERIAL NOT NULL,
    Name VARCHAR(255) NOT NULL,
    DeviceTypeID INT NOT NULL,
    LastConfigTime TIMESTAMP NOT NULL,
    PRIMARY KEY (DeviceID),
    FOREIGN KEY (DeviceTypeID) REFERENCES TSO.DeviceType (DeviceTypeID)
);
```

C.6. SENSINGDEVICECONFIGURATION TABLE CREATION

```
CREATE TABLE TSO.SensingDeviceConfiguration (
    ConfigID SERIAL NOT NULL,
    DeviceID INT NOT NULL,
    ActivatedTime TIMESTAMP NOT NULL,
    PRIMARY KEY (ConfigID),
    UNIQUE (DeviceID, ActivatedTime),
    FOREIGN KEY (DeviceID) REFERENCES TSO.SensingDevice (DeviceID)
);
```

C.7. SENSOR TABLE CREATION

```
CREATE TABLE TSO.Sensor (
    SensorID SERIAL NOT NULL,
    ConfigID INT NOT NULL,
    SensorTypeID INT NOT NULL,
    Position VARCHAR(255) NOT NULL,
    PRIMARY KEY (SensorID),
    UNIQUE (ConfigID, SensorTypeID, Position),
    FOREIGN KEY (ConfigID) REFERENCES TSO.SensingDeviceConfiguration (ConfigID),
    FOREIGN KEY (SensorTypeID) REFERENCES TSO.SensorType (SensorTypeID)
);
```

C.8. SENSORACTIVITY TABLE CREATION

```
CREATE TABLE TSO.SensorActivity (
    SensorID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    Active BOOLEAN NOT NULL,
    Interval INT NOT NULL,
    PRIMARY KEY (SensorID, Time),
    FOREIGN KEY (SensorID) REFERENCES TSO.Sensor (SensorID)
);
```

C.9. HUB TABLE CREATION

```
CREATE TABLE TSO.Hub (
    HubID SERIAL NOT NULL,
    Name CHAR(127) NOT NULL,
    DeviceTypeID INT NOT NULL,
    PRIMARY KEY (HubID),
    FOREIGN KEY (DeviceTypeID) REFERENCES TSO.DeviceType (DeviceTypeID)
);
```

C.10. SENSINGDEVICELOCATION TABLE CREATION

```
CREATE TABLE TSO.SensingDeviceLocation (
    LocationID SERIAL NOT NULL,
    EnvID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    DeviceID INT NOT NULL,
    LocationX REAL NOT NULL,
    LocationY REAL NOT NULL,
    LocationZ REAL NOT NULL,
    PRIMARY KEY (LocationID),
    UNIQUE (DeviceID, Time),
    FOREIGN KEY (EnvID) REFERENCES TSO.Environment (EnvID),
    FOREIGN KEY (DeviceID) REFERENCES TSO.SensingDevice (DeviceID)
);
```

C.11. MEASUREMENT TABLE CREATION

```
CREATE TABLE TSO.Measurement (
    SensorID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    LocationID INT NOT NULL,
    HubID INT NULL,
    Value DOUBLE PRECISION NOT NULL,
    Tag VARCHAR(3) NOT NULL,
    PRIMARY KEY (SensorID, Time),
    FOREIGN KEY (SensorID) REFERENCES TSO.Sensor (SensorID),
    FOREIGN KEY (LocationID) REFERENCES TSO.SensingDeviceLocation (LocationID),
    FOREIGN KEY (HubID) REFERENCES TSO.Hub (HubID)
);
```

C.12. ENVIRONMENT TUPLE CREATION

```
INSERT INTO TSO.Environment
(timezone, name, latitude, longitude, elevation)
VALUES
('GMT+1', 'IBM_CAS_Netherlands', 0, 0, 0);
```

C.13. DROP ALL TABLES

```
DROP TABLE TSO.Measurement;
DROP TABLE TSO.SensorActivity;
DROP TABLE TSO.Sensor;
DROP TABLE TSO.SensorType;
DROP TABLE TSO.SensingDeviceConfiguration;
DROP TABLE TSO.SensingDeviceLocation;
DROP TABLE TSO.SensingDevice;
DROP TABLE TSO.Hub;
DROP TABLE TSO.DeviceType;
DROP TABLE TSO.Environment;
```

C.14. DELETE ALL CONTENT

```
DELETE FROM TSO.Measurement;
DELETE FROM TSO.SensorActivity;
DELETE FROM TSO.Sensor;
DELETE FROM TSO.SensorType;
DELETE FROM TSO.SensingDeviceConfiguration;
DELETE FROM TSO.SensingDeviceLocation;
DELETE FROM TSO.SensingDevice;
DELETE FROM TSO.Hub;
DELETE FROM TSO.DeviceType;
DELETE FROM TSO.Environment;
```

C.15. DROP SCHEME

```
DROP SCHEMA TSO;
```

D

DB2 Queries

D.1. Scheme Creation

```
CREATE SCHEMA TSO;
```

D.2. SensorType Table Creation

```
CREATE TABLE TSO.SensorType (
    SensorTypeID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1
    INCREMENT BY 1),
    Model VARCHAR(127) NOT NULL,
    Quantity VARCHAR(127) NOT NULL,
    Unit VARCHAR(127) NOT NULL,
    DefActive SMALLINT NOT NULL,
    DefInterval INT NOT NULL,
    PRIMARY KEY (SensorTypeID),
    UNIQUE (Model, Quantity)
);
```

D.3. DeviceType Table Creation

```
CREATE TABLE TSO.DeviceType (
    DeviceTypeID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1
    INCREMENT BY 1),
    Brand VARCHAR(127) NOT NULL,
    Model VARCHAR(127) NOT NULL,
    MarketingName VARCHAR(127) NOT NULL,
    Category VARCHAR(127) NOT NULL,
    PRIMARY KEY (DeviceTypeID),
    UNIQUE (Brand, Model)
);
```

D.4. ENVIRONMENT TABLE CREATION

```
CREATE TABLE TSO.Environment (
    EnvID INT NOT NULL GENERATED BY DEFAULT AS IDENTITY (START WITH 1
    INCREMENT BY 1),
    Timezone VARCHAR(127) NOT NULL,
    Name VARCHAR(255) NOT NULL,
    Latitude DOUBLE PRECISION NOT NULL,
    Longitude DOUBLE PRECISION NOT NULL,
    Elevation DOUBLE PRECISION NOT NULL,
    PRIMARY KEY (EnvID)
);
```

D.5. SENSINGDEVICE TABLE CREATION

```
CREATE TABLE TSO.SensingDevice (
    DeviceID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1
    INCREMENT BY 1),
    Name VARCHAR(255) NOT NULL,
    DeviceTypeID INT NOT NULL,
    LastConfigTime TIMESTAMP NOT NULL,
    PRIMARY KEY (DeviceID),
    FOREIGN KEY (DeviceTypeID) REFERENCES TSO.DeviceType (DeviceTypeID)
);
```

D.6. SENSINGDEVICECONFIGURATION TABLE CREATION

```
CREATE TABLE TSO.SensingDeviceConfiguration (
    ConfigID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
    DeviceID INT NOT NULL,
    ActivatedTime TIMESTAMP NOT NULL,
    PRIMARY KEY (ConfigID),
    UNIQUE (DeviceID, ActivatedTime),
    FOREIGN KEY (DeviceID) REFERENCES TSO.SensingDevice (DeviceID)
);
```

D.7. SENSOR TABLE CREATION

```
CREATE TABLE TSO.Sensor (
    SensorID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
    ConfigID INT NOT NULL,
    SensorTypeID INT NOT NULL,
    Position VARCHAR(255) NOT NULL,
    PRIMARY KEY (SensorID),
    UNIQUE (ConfigID, SensorTypeID, Position),
    FOREIGN KEY (ConfigID) REFERENCES TSO.SensingDeviceConfiguration (ConfigID),
    FOREIGN KEY (SensorTypeID) REFERENCES TSO.SensorType (SensorTypeID)
);
```

D.8. SENSORACTIVITY TABLE CREATION

```
CREATE TABLE TSO.SensorActivity (
    SensorID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    Active SMALLINT NOT NULL,
    Interval INT NOT NULL,
    PRIMARY KEY (SensorID, Time),
    FOREIGN KEY (SensorID) REFERENCES TSO.Sensor (SensorID)
);
```

D.9. HUB TABLE CREATION

```
CREATE TABLE TSO.Hub (
    HubID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
    Name CHAR(127) NOT NULL,
    DeviceTypeID INT NOT NULL,
    PRIMARY KEY (HubID),
    FOREIGN KEY (DeviceTypeID) REFERENCES TSO.DeviceType (DeviceTypeID)
);
```

D.10. SENSINGDEVICELOCATION TABLE CREATION

```
CREATE TABLE TSO.SensingDeviceLocation (
    LocationID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1
    INCREMENT BY 1),
    EnvID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    DeviceID INT NOT NULL,
    LocationX REAL NOT NULL,
    LocationY REAL NOT NULL,
    LocationZ REAL NOT NULL,
    PRIMARY KEY (LocationID),
    UNIQUE (DeviceID, Time),
    FOREIGN KEY (EnvID) REFERENCES TSO.Environment (EnvID),
    FOREIGN KEY (DeviceID) REFERENCES TSO.SensingDevice (DeviceID)
);
```

D.11. MEASUREMENT TABLE CREATION

```
CREATE TABLE TSO.Measurement (
    SensorID INT NOT NULL,
    Time TIMESTAMP NOT NULL,
    LocationID INT NOT NULL,
    HubID INT NULL,
    Value DOUBLE PRECISION NOT NULL,
    Tag VARCHAR(3) NOT NULL,
    PRIMARY KEY (SensorID, Time, Tag),
    FOREIGN KEY (SensorID) REFERENCES TSO.Sensor (SensorID),
    FOREIGN KEY (LocationID) REFERENCES TSO.SensingDeviceLocation (LocationID),
    FOREIGN KEY (HubID) REFERENCES TSO.Hub (HubID)
);
```

D.12. ENVIRONMENT TUPLE CREATION

```
INSERT INTO TSO.Environment
(timezone, name, latitude, longitude, elevation)
VALUES
('GMT+1', 'IBM_CAS_Netherlands', 0, 0, 0);
```

D.13. DROP ALL TABLES

```
DROP TABLE TSO.Measurement;
DROP TABLE TSO.SensorActivity;
DROP TABLE TSO.Sensor;
DROP TABLE TSO.SensorType;
DROP TABLE TSO.SensingDeviceConfiguration;
DROP TABLE TSO.SensingDeviceLocation;
DROP TABLE TSO.SensingDevice;
DROP TABLE TSO.Hub;
DROP TABLE TSO.DeviceType;
DROP TABLE TSO.Environment;
```

D.14. DELETE ALL CONTENT

```
DELETE FROM TSO.Measurement;
DELETE FROM TSO.SensorActivity;
DELETE FROM TSO.Sensor;
DELETE FROM TSO.SensorType;
DELETE FROM TSO.SensingDeviceConfiguration;
DELETE FROM TSO.SensingDeviceLocation;
DELETE FROM TSO.SensingDevice;
DELETE FROM TSO.Hub;
DELETE FROM TSO.DeviceType;
DELETE FROM TSO.Environment;
```

D.15. DROP SCHEME

```
DROP SCHEMA TSO;
```

E

Cost Analysis

As mentioned in the requirements (see 2.4.3), a cost analysis is requested by the client. A particular interest is for the low-cost sensor node, as the smartphone application does not induce additional deployment costs (except uploading the application). It is necessary that the cost includes supporting infrastructure.

In the cost analysis of the demonstrator, the costs of the server, the hubs, and the sensor node itself are taken into account. The cost of electricity is not taken into account, with exception of the batteries of the sensor node. The estimation assumes that four AA batteries producing 5V can power the device for a week (which is a rather low bound). Because most of the electronics for the sensor node were ordered abroad, the dollar is used as currency.

Bluemix: *$0*
IBM itself is responsible for the upkeep of their servers, and services they provide (e.g. database, web server). Given the relatively small size of the demonstrator, its usage is negligible in cost.

Sensor Node: *$17.41*
The sensor node consists out of the Arduino board, a breadboard, sensors, wiring, and batteries (schematic in figure 4.1). The cost estimation for each is as follows :

- 4 AA batteries: $1.28[1]
- Breadboard and wiring: $3.98[2]
- Sensors: $8.67[3]
- Arduino Nano: $3.48[4]

Hub : *$0*
Initially was planned to use Raspberry PI's in the demonstrator (mentioned in 2.5). During implementation however we stumbled upon a better solution: usage of the laptops of IBM. These also have both Bluetooth and WiFi capabilities, and do not require any investments. Furthermore, due to network restrictions at IBM, connecting unknown hardware with the network proved troublesome. These laptops did not need to be purchased, nor is any special hardware needed.

Average low-cost sensor node cost: ***$17.41***
This low average cost surpasses the initial requirements of below $100 (req. L4) and below $80 (req. L5). This

[1] http://www.ebay.com/itm/100-PCS-AA-1-5V-Duracell-Duralock-Alkaline-Batteries-Bulk-Wholesale-Exp-2021-/400740322405 (retrieved 7 June 2015)
[2] http://www.ebay.com/itm/MB-102-830-Solderless-Breadboard-Tie-Points-65Pcs-Jumper-Cable-Wires-Arduino-New-/181603575198 (retrieved 7 June 2015)
[3] As calculated in table 4.1
[4] http://www.ebay.com/itm/MINI-USB-Nano-V3-0-ATmega328P-CH340G-5V-16M-Micro-controller-board-for-Arduino-/281626083826 (retrieved 7 June 2015)

is very promising for our demonstrator, which will comprise out of ±25 sensor nodes. This sums up to a total cost of 25 x $17.41 = $435.25 for the demonstrator.

F

SONARQUBE CODE-STYLE DEVIATIONS

As mentioned in section 6.3.1, Sonarqube was used to manage the quality of the code. Among the points of measurements in Sonarqube, it also includes a check of the code-style according to a specific standard. In our project, we decided to deviate from this standard format and adopt our own self-defined code-style, which is better readable and more line efficient, in our opinion. In this section you can find some pieces of code conform the code-style as would be preferred by Sonarqube and conform the code-style as would be preferred by us. The latter is used throughout the implementation of the system, which resulted in a false negative influence on the results in section 6.3.2. In this appendix we provide the reader with argumentation for valuing our preferred code-style over Sonarqube's. First, in section F.1, we define the categories out of which the deviations consist. Second, in section F.2, we provide the reader with two examples of deviations of these categories.

F.1. CATEGORIES

The differences in both can be categorized into three categories:

- **Comments at the end of lines:** Sonarqube prefers that lines do not end with comments, even though it bloats up the code, especially when importing modules, a less desired result in our opinion.
- **Usage of tabulation characters:** Sonarqube prefers that spaces are used, whereas the quickness and ease of using tab characters is convenient, in our view.
- **Trailing white spaces:** Sonarqube prefers that lines never end with white spaces, which seems to make sense. Though the usage of white spaces at the end of the lines in the form of empty lines was seen as useful by us for specific commenting structures.

F.2. EXAMPLES

In this section we provide the reader with two examples, together showing code-style deviations of all three categories stated in F.1.

One example of the differences between the preferences of Sonarqube and our own code-style is displayed in respectively figure F.1 and F.2. We can see that the trailing comments that are preceded by tabs in lines 4-10 in figure F.2, allow for a more readable code if we compare it to the preferred code-style of Sonarqube, which is displayed in figure F.1.

Another example of deviations between the preferences of Sonarqube and our own code-style is displayed in respectively figure F.3 and F.4. In these images we can find a difference in the last item stated above; trailing white spaces. The trailing white spaces used at line 3 in figure F.4 allow for - in our opinion - more readable code in comparison to the preferred code-style of Sonarqube, which is displayed in figure F.3.

```
////////////////////////////////////
// Include necessary query modules

// DB2 library (to communicate with IBM DB2 database)
var ibmdb = require("ibm_db");

// FORCED-GET queries
module.exports.get = require('./forced-get');

// INSERT queries
module.exports.insert = require('./insert');

// CHECK queries
module.exports.check = require('./check');

// REMOVE queries
module.exports.remove = require('./remove');

// SELECT queries
module.exports.select = require('./select');

// SELECT queries
module.exports.update = require('./update');
```

Figure F.1: Code according to the Sonarqube codestyle

```
////////////////////////////////////
// Include necessary query modules
//
var ibmdb = require("ibm_db");                  // DB2 library (to communicate with IBM DB2 database)
module.exports.get = require('./forced-get');   // FORCED-GET queries
module.exports.insert = require('./insert');    // INSERT queries
module.exports.check = require('./check');      // CHECK queries
module.exports.remove = require('./remove');    // REMOVE queries
module.exports.select = require('./select');    // SELECT queries
module.exports.update = require('./update');    // SELECT queries
```

Figure F.2: Code according to our preferred codestyle

```
 /**
  * Initialize database client
  * @param   databaseClient      Database connection client as generated by gen-dbg
  */
 module.exports.init = function(databaseClient) {
     dbClient = databaseClient;
 };
```

Figure F.3: Code according to the Sonarqube codestyle

```
 /**
  * Initialize database client
  *
  * @param   databaseClient      Database connection client as generated by gen-dbg
  */
 module.exports.init = function(databaseClient) {
     dbClient = databaseClient;
 };
```

Figure F.4: Code according to our preferred codestyle

G

RAW SIG FEEDBACK

G.1. FIRST EVALUATION

Reviewer: Dennis Bijlsma (d.bijlsma@sig.eu)
Language: Dutch

De code van het systeem scoort ruim vier sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Interfacing en Unit Size.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Wat binnen dit systeem opvalt is dat er in de C++ code een relatief groot aantal methoden zijn die als laatste parameter een boolean krijgen. Deze wordt vervolgens gebruikt om eventueel een komma toe te voegen aan het resultaat. Hierdoor moet de aanroeper al weten of er nog meer data komt. Een andere opzet hiervoor is om de methode een komma toe te laten voegen als er al een element was. Hierdoor zou de interface eenvoudiger en makkelijker te gebruiken worden. Het is aan te raden dit soort overwegingen mee te nemen in het ontwerp van dit soort API's.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'handle'-methode in de 'handle-sensing-device-identification' module, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld '// Check if a DeviceID is specified and is valid' en '// Make sure the device has a location' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Het opsturen van zowel de schone versie als de versie met libraries geeft trouwens een goed beeld van het project. Het is wellicht goed om in de versie met libraries ook duidelijk te documenteren welke stukken code aangepast dienen te worden, en welke stukken code onderhouden worden door een externe partij. Dit zal toekomstige ontwikkelaars ook helpen om deze zaken uit elkaar te houden.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

G.2. SECOND EVALUATION

Reviewer: Dennis Bijlsma (d.bijlsma@sig.eu)
Language: Dutch

In de tweede upload zien we dat het codevolume is gegroeid, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven.
Op het niveau van de deelscores zien we een soortgelijk beeld als bij de eerste upload: Unit Size en Unit Interfacing, die in de analyse van de eerste upload als verbeterpunt werden aangemerkt, scoren nog steeds het laagste. Op beide aspecten zien we wel een kleine verbetering, maar dit heeft geen significante invloed op de score gehad.
Uit deze observaties kunnen we concluderen dat het systeem weliswaar bovengemiddeld onderhoudbaar is, maar dat de aanbevelingen van de vorige evaluatie beperkt zijn meegenomen in het ontwikkeltraject.

H

INFOSHEET

H.1. THE PROJECT

Project Title: The Sensitive Office (TSO)
Client Organization: IBM Center for Advanced Studies (IBM CAS)
Final Presentation: Monday July 6, 2015 at 13:30 PM

Description:
IBM CAS approached the TU Delft to create a demonstrator for a system that can measure environmental quantities (e.g. temperature), and collect the data that is produced by these measurements in a specific context. During the research phase, we evaluated existing solutions and studied literature pertaining to such measurement systems. We extensively interviewed and discussed possibilities with the client. One of the biggest challenges was the ambitious nature of the project, and the team's initial inexperience with the hardware. These challenges were overcome through responsible prioritization in collaboration with the stakeholders in combination with a focus on iterative development. A framework for the solution was designed, which can be implemented independent of platform, and is currently pending to be patented. The validity of this framework was then demonstrated by our own implementation. The system has been put under heavy scrutiny throughout development, making use of a variety of unit test and coverage frameworks, third party evaluation and a static code analysis tool. Two successful deployments were performed at IBM CAS, demonstrating the ability of the system. The client will continue to develop the system based on our recommendations for future work, potentially transforming the demonstrator into a commercial product.

H.2. THE PROJECT TEAM

DAAN RENNINGS
Interests: System Integration, Customer Interaction, Smartphone Applications
Roles: Lead Documentation, Developer, Quality Assurance, External Affairs, Final Presentation

SIMON KASSING
Interests: Information Systems, API Design, Open-source, Online Communities
Roles: Lead Developer, API Designer, Deployment Manager, Repository Manager, Documenter

H.3. CONTACT INFORMATION

Team Members: Daan Rennings (d.j.a.rennings@gmail.com)
Simon Kassing (s.a.kassing@gmail.com)
Client: Drs. Robert-Jan Sips (*IBM Center for Advanced Studies, Amsterdam*)
TU Coach: Prof. dr. ir. Geert-Jan Houben (*SCT Department, Web Information Systems Group*)

The final report for this project can be found at: `http://repository.tudelft.nl`

BIBLIOGRAPHY

[1] P. R. Carlile, *Transferring, translating, and transforming: An integrative framework for managing knowledge across boundaries,* Organization science **15**, 555 (2004).

[2] T. Mens, *Introduction and roadmap: History and challenges of software evolution* (Springer, 2008).

[3] L. Atzori, A. Iera, and G. Morabito, *The internet of things: A survey,* Computer networks **54**, 2787 (2010).

[4] P. Brody and V. Pureswaran, *The next digital gold rush: how the internet of things will create liquid, transparent markets,* Strategy & Leadership **43**, 36 (2015).

[5] M. Dunnewind, S. A. Kassing, D. J. A. Rennings, and G. M. A. Wassenaar, *Assessing network architectures for sensor networks of ubiquitous devices,* (2015).

[6] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, *Internet of things (iot): A vision, architectural elements, and future directions,* Future Generation Computer Systems **29**, 1645 (2013).

[7] R. J. Sips, A. Bozzon, G. Smit, and G. J. P. M. Houben, *The inclusive enterprise: Vision and roadmap,* ICWE 2015 (submitted) (2015).

[8] R. Z. Goetzel, R. J. Ozminkowski, L. I. Sederer, and T. L. Mark, *The business case for quality mental health services: why employers should care about the mental health and well-being of their employees,* Journal of Occupational and Environmental Medicine **44**, 320 (2002).

[9] R. J. Sips, W.-J. Man, B. Havers, and G. J. Keizers, *The poseidon project: An introduction,* 3rd United Nations World forum on Electronic Government .

[10] A. Overeem, J. R Robinson, H. Leijnse, G.-J. Steeneveld, B. P Horn, and R. Uijlenhoet, *Crowdsourcing urban air temperatures from smartphone battery temperatures,* Geophysical Research Letters **40**, 4081 (2013).

[11] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, *A survey on sensor networks,* Communications magazine, IEEE **40**, 102 (2002).

[12] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, *Wireless sensor networks for habitat monitoring,* in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications* (ACM, 2002) pp. 88–97.

[13] A. Milenković, C. Otto, and E. Jovanov, *Wireless sensor networks for personal health monitoring: Issues and an implementation,* Computer communications **29**, 2521 (2006).

[14] M. Srivastava, T. Abdelzaher, and B. Szymanski, *Human-centric sensing,* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **370**, 176 (2012).

[15] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan, *Medusa: A programming framework for crowd-sensing applications,* in *Proceedings of the 10th international conference on Mobile systems, applications, and services* (ACM, 2012) pp. 337–350.

[16] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma, *Prism: platform for remote sensing using smartphones,* in *Proceedings of the 8th international conference on Mobile systems, applications, and services* (ACM, 2010) pp. 63–76.

[17] C. Campolo, A. Iera, A. Molinaro, S. Y. Paratore, and G. Ruggeri, *Smartcar: An integrated smartphone-based platform to support traffic management applications,* in *Vehicular Traffic Management for Smart Cities (VTM), 2012 First International Workshop on* (IEEE, 2012) pp. 1–6.

[18] ISO/IEC 25010:2011 (en), *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, Standard (International Organization for Standardization, Geneva, CH, 2011).

[19] ISO/IEC 9126-1:2001 (en), *Software engineering – Product quality – Part 1: Quality model*, Standard (International Organization for Standardization, Geneva, CH, 2001).

[20] E. S. Yu, *Towards modelling and reasoning support for early-phase requirements engineering,* in *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on* (IEEE, 1997) pp. 226–235.

[21] K. Brennan *et al.*, *A guide to the business analysis body of knowledge (Babok Guide)* (International Institute of business analysis, 2009).

[22] L. Rising and N. S. Janoff, *The scrum software development process for small teams,* IEEE software **17**, 26 (2000).

[23] A. M. Davis, *Operational prototyping: A new development approach,* Software, IEEE **9**, 70 (1992).

[24] D. Graham, E. Van Veenendaal, and I. Evans, *Foundations of software testing: ISTQB certification* (Cengage Learning EMEA, 2008).