Extensible data acquisition tool for Android

Demonstrated with a prototype for sleep monitoring

Svein Petter Gjøby
Master’s Thesis Spring 2016
Abstract

Mobile devices become increasingly more powerful, and can now connect to a variety of external sensors. However, different external sensors might use different communication channels and data protocols, which makes the use of different sensors burdensome. In this thesis, we propose a data acquisition system for Android to make application development easier. The proposed system hides the low-level sensor specific details from the application developer by separating the software into two components, i.e. providers and sensor wrappers. A sensor wrapper is created for a specific sensor supported by the system, and enables any application to use it to collect data from that sensor. The abstraction simplifies the development of applications (i.e. provider applications) that rely on data collected by external sensors by creating a interface that can control data acquisition with any of the sensor wrapper applications. Additionally, the abstraction reduces the amount of duplicate code by implementing sensor specific code once, and reuse the implementation across applications. As a part of the thesis a prototype sensor wrapper and provider application is created to show that the system is extensible, demonstrating that it can support current and future sensors. The created sensor wrapper is adapted to collect data from a biomedical sensor board named BITalino in a sleep monitoring scenario. Several experiments are performed to show that the implemented prototype is stable, resilient, suited for data acquisition during long periods where the device is stationary (i.e. sleep monitoring), and has moderate resource usage. The results show that the prototype developed in this thesis provides stable data acquisition from the BITalino sensor board, while making it feasible to perform further computation with the collected data by keeping the resource usage low.
Acknowledgements

First and foremost, I would like to give a special thanks to my supervisors Thomas Peter Plagemann and Stein Kristiansen for their guidance throughout this thesis. They have provided invaluable insight, and this thesis would not have been possible without their guidance and help. The door to Prof. Plagemann’s office was always open whenever I ran into a trouble spot, or merely wanted to discuss a topic. For that I am grateful. I have been fortunate to make many new friends during my time at the University of Oslo. There are too many to mention, you know who you are. Finally, I would like to thank my parents and sisters for always supporting me, and Pernille for providing me unfailing support and encouragement through the process of writing this thesis.
Contents

1 Introduction and Background 17

1 Introduction 19
   1.1 Motivation ............................................. 19
   1.2 Problem statement ..................................... 21
   1.3 Contributions ......................................... 23
   1.4 Thesis structure ...................................... 24

2 Physiological computing 27
   2.1 Physiological computing alternatives ................. 27
      2.1.1 Shimmer ........................................... 28
      2.1.2 BITalino ......................................... 28
      2.1.3 E-Health Sensor Shield .......................... 28
      2.1.4 Comparison ....................................... 29
   2.2 BITalino .............................................. 30
      2.2.1 History ........................................... 30
      2.2.2 Overview ......................................... 30
      2.2.3 Micro-controller unit ............................. 31
      2.2.4 Data packets .................................... 32
      2.2.5 Sensors and interpreting the data ................ 33
      2.2.6 Java SDK ......................................... 34

3 Android OS 39
   3.1 Android architecture .................................. 39
   3.2 Application Components ............................... 40
      3.2.1 Activities ....................................... 40
      3.2.2 Services ......................................... 42
      3.2.3 Broadcast Receivers .............................. 42
   3.3 Processes and threads ................................ 42
      3.3.1 Process’s lifecycle ............................... 43
      3.3.2 Threads .......................................... 43
   3.4 Interprocess communication ............................ 44
   3.5 Files .................................................... 45
      3.5.1 Internal storage .................................. 45
      3.5.2 External storage .................................. 46
      3.5.3 File size ......................................... 46
   3.6 Energy saving ......................................... 46
II Design and implementation

4 Analysis and High-level design

4.1 Requirement Analysis

4.2 High-level design

4.2.1 Protocol layer solution

4.2.2 Modular solution

4.3 Data

4.3.1 Requirements

4.3.2 Data format

4.3.3 Data packets

4.4 Separation of concern

4.4.1 Linked layer

4.4.2 Data exchange protocol

4.4.3 Using the collected data

4.4.4 Sensor wrapper management

4.5 Provider design

4.5.1 Sensor wrapper management

4.5.2 Sensor wrapper discovery

4.5.3 Data processing

4.5.4 States

4.6 Sensor wrapper design

4.6.1 Adapting to the data source

4.6.2 Configurations

4.6.3 Forwarding the collected data

4.6.4 States

4.7 Combing provider and sensor wrapper applications

5 Android specific design and implementation

5.1 Interprocess communication

5.1.1 Interface between applications

5.1.2 Comparing alternatives

5.1.3 Defining an AIDL Interface

5.1.4 Sending and receiving broadcasts

5.1.5 Broadcast receivers

5.2 Android components

5.3 Sensor wrapper discovery

5.4 Data acquisition

5.4.1 Starting the data acquisition

5.4.2 Stopping the data acquisition

5.4.3 Data flow

5.5 Provider application

5.5.1 ListActivity

5.5.2 SettingsActivity

5.5.3 CollectActivity

5.6 BITalino sensor wrapper application

5.6.1 Development with BITalino

5.6.2 User interface
A.2 Adapting to the data source ........................................ 147
A.2.1 Sending metadata ........................................... 148
A.2.2 Send metadata ............................................... 148
A.2.3 Connecting to the data source .............................. 148
A.2.4 Collecting data from the data source .................... 150
A.2.5 DataHandler ................................................ 151
A.2.6 Closing connection ......................................... 152
A.3 Configuration ................................................... 153
A.3.1 Defining the layout ....................................... 153
A.3.2 Storing the configuration ................................. 153

B Contributions to the BITalino Java API 155
B.1 Analog channel always zero ................................. 155
B.2 Scaling functions ........................................... 156
B.3 Enchantments ................................................ 156
B.3.1 Packet sequence check ................................. 156
B.3.2 Added scaling functions ............................... 157

C User manual for prototype 161
C.1 Turning on the BITalino ...................................... 162
C.2 Initial setup .................................................. 162
C.2.1 Pair the devices over bluetooth ....................... 162
C.2.2 Specify MAC-address of BITalino .................... 162
C.3 Collect data ................................................ 164
C.3.1 Configure BITalino .................................... 164
C.3.2 Collecting ............................................... 165
C.3.3 Configure data storage ............................... 165
C.3.4 Start collecting ....................................... 167
C.4 Access collected data ..................................... 167

D Python scripts 169
D.1 Common features ........................................... 169
D.2 Fall detection ............................................... 171
D.3 Database storage ......................................... 173
D.4 Visualization ............................................... 175
List of Figures

1.1 Illustration of application and sensor support decoupling . 22
2.1 Shimmer3 [16] .................................................. 28
2.2 BITalino Plugged kit [17] ................................ 29
2.3 e-Health Sensor Shield [18] ............................... 29
3.1 Lifecycle of an Android Activity [26] ....................... 41
3.2 Illustration of Binder communication [29] ................. 45
3.3 Illustration of the effects of Doze [35] ....................... 47
4.1 Architecture of a naive solution ............................. 54
4.2 Architecture of a protocol layer solution .................... 55
4.3 Architecture of a software separating solution ............. 56
4.4 Sensor wrapper used by one single application .......... 69
4.5 Sharing the collected data between multiple applications 70
5.1 Android components of the two application types ....... 81
5.2 Sensor wrapper discovery .................................... 83
5.3 Program flow when starting data acquisition ............. 84
5.4 Program flow when stopping data acquisition .......... 86
5.5 Data flow during data acquisition ....................... 88
5.6 The Activities of the provider application ............... 91
5.7 Screenshot of the ListActivity ............................ 92
5.8 Screenshot of the SettingsActivity ....................... 94
5.9 Screenshot of the CollectActivity ....................... 95
5.10 Screenshot of the BITalino Sensor Wrapper ......... 98

6.1 CPU-usage for all runs with a sampling frequency of 1 Hz 118
6.2 CPU-usage for all runs with a sampling frequency of 10 Hz 119
6.3 CPU-usage for all runs with a sampling frequency of 100 Hz ........................................... 119
6.4 CPU-usage for all runs with a sampling frequency of 1000 Hz .............................................. 120
6.5 Comparison of the average CPU-usage of the different sampling frequencies .......................... 121
6.6 Experiment set up .............................................. 122
6.7 Energy consumption per hour (mAh) .................. 127
6.8 Comparison of the average CPU-usage of the different sampling frequencies .......................... 131
6.9 Comparison of the average energy consumption of the different sampling frequencies ............... 132

7.1 Overview of the implemented data acquisition system ... 141

C.1 Application icons of Collector and BITalino ............... 161
C.2 Picture of the BITalino with and without protection case . 162
C.3 Screenshot of the bluetooth configuration ............... 163
C.4 User interface of BITalino application ............... 163
C.5 Backside of a BITalino sensor board ............... 164
C.6 Screenshot of the BITalino application ............... 165
C.7 User interface of Collector application when launched ... 166
C.8 Screenshot of Collector application settings ............... 166
C.9 User interface after selecting sensor wrappers ............... 167
C.10 Screenshot of file explorer ............... 168

D.1 Euclidean length of a vector ............... 171
D.2 Plot of luminous values queried from the database ............... 176
D.3 Screen shot of Kst ............... 178
List of Tables

2.1 State commands ........................................ 32
2.2 Additional commands .................................... 32
2.3 BITalino data packet ..................................... 33

4.1 Illustration of metadata ................................. 61

6.1 Device specifications and settings ..................... 114
6.2 Average statistics of the measured CPU-usage for 1 Hz . 118
6.3 Average statistics of the measured CPU-usage for 10 Hz . 118
6.4 Average statistics of the measured CPU-usage for 100 Hz . 119
6.5 Average statistics of the measured CPU-usage for 1000 Hz 120
6.6 Results from the long initial runs ...................... 124
6.7 Results from the short validation runs ................. 126
6.8 Results of time test .................................... 129
6.9 Results of repetition test ............................... 129
Listings

2.1 Overview of the BITalinoDevice class ............................ 35
2.2 Overview of the BITalinoFrame class .............................. 36
2.3 Overview of the SensorDataConverter class ....................... 37
2.4 Sample code for scaling BITalino data ............................ 37
3.1 Request of permission to write to the external storage ......... 46
4.1 A XML structure describing the encoding of a name ............ 59
4.2 A JSON structure describing the encoding of a name .......... 59
4.3 A JSON structure describing the encoding of the metadata ... 61
4.4 A JSON structure describing the encoding of a data reading 62
5.1 AIDL interface between the two application types ............. 73
5.2 Implementation of the AIDL interface ............................ 74
5.3 Connection handling to the remote Service ...................... 75
5.4 Sending a string to the remote Service .......................... 75
5.5 Manifest declaration of a BroadcastReceiver .................... 76
5.6 Sending an Intent .............................................. 76
5.7 Receiving an Intent ............................................. 77
5.8 Sending a HELLO-command as an Intent .......................... 77
5.9 Sending a REGISTER-command as an Intent ...................... 77
5.10 Sending a START-command as an Intent ......................... 78
5.11 Sending a STOP-command as an Intent .......................... 78
5.12 Implementation of a RespondReceiver ........................... 78
5.13 Implementation of a RegisterReceiver ........................... 79
5.14 Implementation of a StartReceiver .............................. 80
5.15 Implementation of a StopReceiver ............................... 81
5.16 Binding to the provider application ............................. 85
5.17 Creating and starting the communication thread ............... 85
5.18 Telling the WrapperService to stop data acquisition ........ 86
5.19 Creating and starting the communication thread ............... 87
5.20 Creation and sending of JSON-object ........................... 89
5.21 Implementation of a runnable DispatchTCPHandler object .... 89
5.22 Implementation of a runnable DispatchFileHandler object ..... 90
5.23 Code to start the SettingsActivity .............................. 93
5.24 Code to start the CollectActivity with the selected sensor wrappers ........................................... 93
5.25 Code for saving values to SharedPreferences .................. 95
5.26 Starting the thread for communicating with the data source 99
5.27 Main actions of the communication thread ........................................ 99
5.28 Method for creating and sending a metadata packet .......................... 100
5.29 Method for connecting to the BITalino device .................................. 102
5.30 Method for collecting data values from the BITalino board ................. 103
5.31 Method for extracting and sending the collected data ......................... 104
5.32 Method for cleaning up the connections to the data source ................. 104
5.33 Reconnection mechanism for the BITalino sensor wrapper ................. 106
5.34 Acquiring permission to use WakeLock .............................................. 107
5.35 Acquiring a WakeLock ..................................................................... 107
5.36 Setting an Android component to the foreground .............................. 109
5.37 Setting the rank of an Android component back to default ................. 109
6.1 Sample output of CPU measurements using Busybox ......................... 115
6.2 Python script parsing the BusyBox output ......................................... 116
6.3 Python script for detecting packet reordering ................................... 123
A.1 Download and rename the template project ...................................... 145
A.2 Importing template project to Android Studio .................................. 146
A.3 Packet re-factorizing ....................................................................... 146
A.4 Updating the ID of the project .......................................................... 146
A.5 Changing the name of the application .............................................. 147
A.6 Creating and sending metadata ....................................................... 148
A.7 Establishing a connection to a BITalino sensor board ....................... 149
A.8 Collecting data from a BITalino device .......................................... 150
A.9 Changing the constructor of the data-object .................................. 151
A.10 Extracting and sending the collected data ...................................... 152
A.11 Closing the connections to a BITalino sensor board ....................... 152
A.12 Layout definition of sensor wrapper ............................................. 153
A.13 Saving values from the layout-objects ......................................... 154
D.1 Usage message of scripts ............................................................... 169
D.2 Creating a TCP socket ................................................................. 170
D.3 Timeout mechanism of socket ....................................................... 170
D.4 Reading and splitting data into data packets .................................. 171
D.5 Threshold-based fall detection algorithm ...................................... 172
D.6 Helper functions for database script ............................................. 173
D.7 Main loop of the database script ................................................... 174
D.8 Function for saving the metadata to matrices .................................. 174
D.9 Function for saving data to the database ....................................... 175
D.10 Query for obtaining luminosity values ........................................ 175
D.11 Structure of the shared file .......................................................... 176
D.12 Main loop of the plot script ......................................................... 177
D.13 Terminal command to start Kst .................................................... 177
Part I

Introduction and Background
Chapter 1

Introduction

1.1 Motivation

In recent years smart phones have become more and more powerful. With increasing processing power, better battery life, higher network speeds, and larger screens the shift away from personal computers is accelerating. Modern smart phones are delivered with multiple internal sensors, and are capable of connecting to a large range of external sensors using wired and wireless communication channels. These external sensors range all the way from closed commercial packages (of both hardware and software) to open low-cost platforms. On the commercial side of the spectrum there are vendors like Garmin [1], Polar [2] and FitBit [3], which among other things offer activity tracking of the user. These platforms are proprietary and often closed for the outside user. On the other side of the spectrum there are vendors like BITalino [4], Shimmer [5] and Cooking Hacks [6], which enables anyone to monitor and to create interactive programs using physiological values of their own body. They provide low-cost sensors with an open standard which make them easy to use by the end user. The large range of available sensors can increase the utility of smart phones in many areas, and provide great benefits in fields that require extensive monitoring, such as science and medicine. Particularly healthcare services, which previously were bound by stationary medical devices, are being replaced with mobile solutions, which increases the independence of the patient. These applications are a part of a research field called Mobile Health (hereafter referred to as mHealth), which covers "medical and public health practice supported by mobile devices, such as mobile phones, patient monitoring devices, personal digital assistance, and other wireless devices" [7]. mHealth is based on the use of mobile communication for healthcare and well-being services. Ranging all the way from the possibility to call an emergency number for help, to real-time analysis of data (as glucose or pulse measurements) collected and analysed by a mobile device. In high-income countries the development of mHealth is driven by a wish to reduce the cost of healthcare [7]. This can for instance be achieved by using mobile devices for remote patient...
monitoring, which in turn can reduce hospitalization [8]. Remote patient monitoring and other mHealth applications are made possible by using collected medical data in a smart way, such as a continuous analysis of measured glucose levels to determine if the monitored person needs to eat. Earlier the possibilities of mHealth applications were limited by the technology. However, with the capabilities of modern mobile devices the applications are now only limited by the creativity of the developer. A good example of the potential of mHealth applications is the CESAR project, which aims at using low-cost external sensors to supplement classical sleep monitoring to improve diagnosis of obstructive sleep apnea (OSA). "Obstructive sleep apnea is being increasingly recognized as an important cause of medical morbidity and mortality. It is a relatively common sleep disorder that is characterized by recurrent episodes of partial or complete collapse of the upper airway during sleep" [9]. Unfortunately, the process of diagnosing OSA (which involves hospitalization in sleep laboratories) is resource demanding, leaving many patients under-diagnosed and untreated. As an effort to lower the threshold and cost of detecting OSA, the CESAR project aims at developing new software solution using state-of-the-art consumer electronic devices and low-cost sensors to enable anyone to monitor and analyse physiological parameters relevant for OSA monitoring at home. The BITalino sensor kit (including a range of physiological sensors) is a promising technology for collecting data to use in the CESAR project. Currently BITalino supports all except one (a sensors to measure nasal airflow) of the sensors intended to be used in the project. However, this might be supported in the future. The quality of the data provided by the BITalino sensor kit is unexplored. Further studies might reveal that it is not of sufficient quality, and it might therefore be preferred to use other sensors from other vendors. However, changing the data management and analysis part of the software is not desired when changing to another sensor. This illustrates the need of decoupling the data collection and data analysis, making it possible to change either one without affecting the other.

As mentioned earlier, modern mobile devices are capable of connecting to a range of external sensors using wired and wireless network technologies, such as Bluetooth [10], ANT+ [11], ZigBee [12] Wi-Fi [13] etc. Unfortunately, different external sensors might use different Link Layer technologies, and communication protocols, making the implementation of support for different sensors challenging and time consuming. As an effort to address this challenge an extensible data acquisition system for mobile devices, which collects data from arbitrary sensors to either store it or forward it to an application performing analysis on the collected data, is proposed in this thesis. The system should be extensible in the sense that dynamically adding support for new external and internal sensors is seamless, making it possible to reuse the implementation of supported sensors across applications. To evaluate and demonstrate the capabilities and extensibility of the data acquisition
system support for the BITalino sensor kit is added to the system. The extensible data acquisition system, with the support for the BITalino sensor kit, is intended as a data acquisition platform to expedite the CESAR project; making it possible to change to another sensor vendor without affecting the data management and analysis software.

1.2 Problem statement

Today a large range of different external and built-in sensors are available for mobile devices, and chances are that the number of available sensors increase in the future. In general, adding support for an external sensors can be divided into two levels. At the bottom is the communication media (the Link Layer) that allows information and commands to be exchanged between the computing device (the mobile device) and the data source (the sensor). With such a communication connection established the devices need to agree on a common way of representing the interchanged data. Sensor platform manufacturers often provide a low-level protocol for interchanging data, see Section 3.3 for the protocol used by BITalino. These low-level protocols are often based on sending minimal byte sequences between the communication entities. To ease the usage of these protocols it is a common practice to develop programming language specific software development kits (SDKs), which hide the details of the low-level protocols. Unfortunately, different external sensors might use different Link Layer technologies and low-level communication protocols for communicating with the sensors. Different Link Layer technologies (such as Ethernet, USB, Bluetooth, WiFi, ANT+ and ZigBee) imply different ways of establishing a connection. Bluetooth devices need to be paired and connected, while over WiFi devices can address each other directly without pairing. Additionally, the low-level communication protocols from different sensor manufacturers are not standardized, neither are the SDKs that hide the details of these protocols; they might expose different commands and methods. This makes it burdensome to add support for a new sensor, it requires a lot of insight into the sensors specific details. An application that manages or analyses the collected data (hereafter known as an app) might integrate the support for the sensors within the application itself. Moreover, establishes a connection and implements the low-level protocol with the sensor. This introduces duplicate work and code if multiple apps uses the same sensor. Imagine three different apps, which all use the same sensor; they all implement the connection and communication with that sensor. The support for the sensor is added three times, instead of only once. This duplicate work could be avoided by doing exactly that, implementing the support of the sensor once and reuse this implementation across apps, as shown in Figure 1.1. The re-usage of sensor support across apps yields the need for one common interface for controlling any of the sensors. This hides the technical low-level details of the sensors from the developer creat-
Figure 1.1: Illustration of application and sensor support decoupling

The apps, and they only need to relate to one interface. A framework that isolates the sensor support and the apps might make it easier to develop apps that utilize the collected data. Additionally, isolating the sensor support into modules makes it easier to test the code, which improves the robustness and quality of the implementation. As mentioned earlier, different sensors use different representations of their collected data; standardizing the data representation is necessary in order to create one common interface. However, different sensors might measure different physical phenomena. This is a contrast in relation to the isolation of the sensor support, which makes the environment and situation (hereafter referred to as the context) of the collected data unclear. To cope with this, the data representation needs to inform the apps of the context of each sensor (what they measure and which unit it is measured in). Another benefit of using a common interface is that apps easily can exchange from using one data source (sensors) to using another data source without much effort. As for the CESAR project, where the researchers might discover that the data quality delivered by BITalino is not good enough, and therefore want to explore the capabilities of other sensors. In this thesis we aim at designing and developing an extensible system, which enables apps to collect data from various external and built-in sensors through one common interface. The design should be valid for all Unix/Linux based system as far as possible, increasing the portability to other platforms. However, the focus for this thesis is to design and implement a system for Android. The architecture of Android facilitates for multiple approaches of isolating the sensor support, some of which involve interprocess communication. The choice of an architectural approach introduces tradeoffs between modularity and performance; increasing modularity might decrease performance. Increased modularity makes it easier to deploy the sensor support. However, increasing the modularity might affect the performance negatively. Unlike traditional computers, the battery power is limited on mobile devices. In order to give the user the best possible user experience the operating system provides multiple power saving mechanisms to increase the lifetime of mobile devices. To achieve this, applications and features of the mobile device might be temporarily turned off to save power, which might make data acquisition impossible.
Therefore, these mechanisms need to be taken into account when creating the data acquisition system, to enable stable data acquisition even over a long period.

The challenges presented previously in this section provide both functional and nonfunctional requirements to the acquisition system, which is further discussed in Section 5.1. Thus, first a requirements analysis is done to determine the needs of an extensible data acquisition system. After the requirement analysis a design, implementation and evaluation method is used to give an overall understanding what needs to be done, how it is carried out, and how it is evaluated.

1.3 Contributions

By addressing the challenges described in the problem statement multiple contributions are made in this thesis. Firstly, a software design for decoupling the data collecting from the data management and data analysis is proposed. The design is as far as possible independent of the Android platform, and the design describes the data acquisition system with a common low-level interface for managing and collecting data from an arbitrary sensors. The design introduces an abstraction called a Sensor Wrapper, which is the support of a specific sensor (or data source). A prototype of the data acquisition system is implemented for Android based on the proposed design. The prototype is capable of collecting data from both external and built-in sensors, which then is either saved to a file or sent to a remote server over the internet. Moreover, Sensor Wrappers are created for both the BITalino sensor kit and the built-in accelerometers. The BITalino sensor kit was integrated in the system to expedite the CESAR project. It is aiming at performing sleep monitoring with a mobile device at home during a whole night, and possibly perform real-time analysis of the collected data on the mobile device. A range of experiments are conducted to show that the developed prototype resolves the challenges described in the problem statement. Moreover, it provides the required stability in order to collect data during a whole night. Additional experiments were also conducted to show that the prototype has moderate resource usage. Making it feasible to perform real-time analysis of the collected data on the mobile device itself. Furthermore, the design proposes a standardization for representing the collected data and gives a description of the common interface for using the sensors. To make it easy to integrate support for a new data source (sensor) in the system, a template Sensor Wrapper project for Android is provided together with a detailed description of the required implementation.
1.4 Thesis structure

In this thesis, a prototype of an extensible data acquisition system for Android is developed. The prototype stores the collected data to a file or sends it from the phone, and is extensible to seamlessly add support for new data sources. Moreover, support for a biomedical sensor board named BITalino is integrated in the system. The thesis is divided into four parts and 9 chapters.

The motivation for the thesis is described in the first chapter of Part 1, and stems from the goal of separating the data acquisition from the application using the collected data. To decrease the amount of duplicate work and code when collecting data from a sensor by enabling data collection from arbitrary sensors through one common interface. Additionally, integrating support for the BITalino sensor board is motivated by a goal to create a data acquisition platform to expedite the CESAR project. Chapter 2 begins with a description of various biomedical sensor boards, their capabilities and shortcomings in a sleep monitoring scenario are discussed. Then the BITalino sensor board is described in detail. At most six analog sensors can be connected to the BITalino sensor board at the same time, the value from each of the connected sensors are sampled 1, 10, 100 or 1000 times each second when collecting data with the BITalino sensor board. Lastly in Part 1, Chapter 3 serves as an introduction to the Android operating system.

In Part 2, a requirement analysis is done in Chapter 4, to determine the needs of the extensible data acquisition system. Based on the requirements found in the analysis a software separation design is proposed. The proposed separation is into two application types, a sensor wrapper application responsible for the sensor specific integration and data collection, and a provider application responsible for managing the sensor wrapper applications and for processing the data received from the sensor wrappers. The Android independent part of the design is also specified in Chapter 4. This involves standardizing the representation of the collected data, and assigning responsibilities to each of the two applications. The Android specific design choices are described in Chapter 5. An interprocess communication mechanism is chosen and a low-level interface between the applications is defined based on this mechanism. The provider application uses this interface to discover, receive the collected data, start and stop the data acquisition from the sensor wrappers. The chapter further describes various details of the implementation of the prototype.

In Chapter 6, a series of experiments are described. These experiments are performed to show that the implemented prototype satisfies the requirements presented in the requirement analysis. The key results from the experiments are presented and discussed. Lastly, in Chapter 7 the developed prototype is discussed and evaluated, before
the open problems and future work is discussed.
Chapter 2

Physiological computing

Physical computing deals with the study and development of interactive systems that sense and react to the analog world. One of the fields within physical computing is physiological computing, which deals with the study and development of systems that sense and react to the human body. In this chapter an overview over some physiological computing alternatives are presented and compared. Then, the BITalino sensor kit is described in more detail.

2.1 Physiological computing alternatives

Physical computing has open sensor platforms like Arduino [14] (launched in 2005), which enables anyone to develop interactive physical systems. Physical computing is typically used to detect primitive events (like movement), while physiological computing is used to detect more delicate events (like pulse or the oxygen saturation in the blood). Naturally, physiological computing requires more accurate data acquisition than physical computing; more accurate in terms of both the accuracy of the sampling rate (for instance to correctly determine the pulse) and the measured values (for instance to accurately determine the oxygen saturation). The requirements of higher accuracy makes low-cost physiological sensors hard to obtain. In 2004, the OpenEEG project [15] was launched with a mission to offer a low-cost physiological platform allowing hobbyist to monitor brain activity by collecting electroencephalography (EEG) signals. Since 2004 multiple open physiological sensor platforms have emerged, to offer a wider range of physiological sensors at a low-cost. Among these are Shimmer, BITalino and Cooking Hacks’ e-Health Sensor platform, which are introduced in the subsequent section. First, an overview of the three platforms is given. Then their capabilities and relevance to the CESAR project are discussed.
2.1.1 Shimmer

Shimmer [5] offers a wearable sensor platform. It includes basic motion sensing (including accelerometer, gyroscope, magnetic sensors and an altimeter), a programmable micro-controller to set up data capture, bluetooth for wireless transmission, internal storage for data, and a rechargeable battery. Additionally, it is possible to purchase free-standing units measuring electrodetermal activity (EDA), electrocardiography (ECG) and electromyography (EMG) which can be connected to the sensor platform over a wireless medium. To ease the integration with other software Shimmer offer programming APIs for Java, C# and labView.

![Figure 2.1: Shimmer3 [16]](image)

2.1.2 BITalino

“BITalino is a low-cost toolkit to learn and prototype applications using body signals” [4]. It offers three different sensor kits, all of which consists of a programmable micro-controller unit, a Bluetooth unit for transmission, and a rechargeable battery. A wide range of both physiological and physical sensors are offered by the BITalino team: light, temperature, accelerometer, EDA, ECG, EMG, pulse sensor, electroencephalography (EEG), oxygen saturation and respiration. The available sensors can be connected to the sensor kit with six analog inputs. However, the user is free to combine sensors and analog inputs as desired, with a maximum of six sensors connected simultaneously (limited by the number of analog inputs). Multiple programming SDKs (APIs) for multiple programming languages (such as Java, C++, labView, Matlab, Python) and tutorials are available at the BITalino web page to make development easier.

2.1.3 E-Health Sensor Shield

Cooking Hacks’ e-Health Sensor platform [6] offers sensors only expected to be seen at a hospital. The e-Health Sensors Shield has to be used in combination with either an Arduino or a Raspberry Pi to perform body monitoring with ten different sensors: pulse,
Figure 2.2: BITalino Plugged kit [17]

oxygen saturation, nasal airflow, body temperature, glucosometer, blood pressure, accelerometer, EDA and EMG. The shield does not contain a dedicated micro-controller and is dependant on either an Arduino or a Raspberry Pi for reading and forwarding the values from the shield. A c++ library is offered to let the developer easily read the values from the shield and send it using any of the network technologies available offered for Arduino and Raspberry Pi.

Figure 2.3: e-Health Sensor Shield [18]

2.1.4 Comparison

All of the three above mentioned sensor platforms are open physiological platforms. However, some more suited for the CESAR project than others. The CESAR project aims at lowering the threshold for performing sleep monitoring to detect obstructive sleep apnea (OSA) by making the monitoring tools more available. In order to do this the costs need to be kept low, the sensing device needs to be portable and as small as possible. To determine if the monitored person has OSA certain sensors is used: respiration sensors placed around the chest and stomach, a nasal airflow sensor, an oxymeter sensor to measure the oxygen saturation
in the blood and possibly an EEG sensor to determine if the person is sleeping.

The Shimmer Engineering team has a great focus on providing accuracy and high quality of the collected signals, this is done by selecting the best possible components. Unfortunately, this results in a significantly higher retail price than both the BITalino sensor kit and the e-Health sensor shield. Unlike Shimmer their focus is to deliver acceptable quality at a lower cost. Additionally, Shimmer does not offer any of the desired sensors for the CESAR project, making it unsuited for the project. Cooking Hacks’ e-Health sensor shield offers all the desired sensors. However, it needs to be used in combination with either an Arduino or a Raspberry Pi, which both are dependant on a charging cable out of the box. This makes it undesirably large and inflexible. The BITalino sensor kit offers almost all of the desired sensors (except nasal airflow), is small and battery powered, and offers a lot of flexibility in terms of combining various sensors. Additionally, multiple well documented programming interfaces are available, making it the desired starting point for the CESAR project.

2.2 BITalino

This section serves as an introduction to the BITalino sensor kit and is based on the documentation found on the BITalino website [4]. It starts with the historical aspect and a general overview of the anatomy of the BITalino, followed by a more detailed description of the micro-controller unit and the various sensors. The BITalino is distributed as three different kits, differentiated by how the sensors are connected to the device. The focus of this introduction is the BITalino Plugged Kit, which has a pluggable cable connection between the sensors and the BITalino sensor board.

2.2.1 History

BITalino is a company that was created in 2007 as a joint work between a commercial company named PLUX and the academic institution IT-Institute de Telecomunicações, both of which are located in Portugal. The company specializes in creating innovative biomedical hardware and software for healthcare and research. The vision of the founders of BITalino is to make bio-signals available for anyone at a low-cost.

2.2.2 Overview

The BITalino sensor kit consists of a range of interchangeable sensors, a programmable micro-controller unit, a LED light for visual feedback, a Bluetooth unit for communication, a power management block for giving power to the different components, and a rechargeable battery. The BITalino device connects with another computer device (such as a smartphone or a laptop) over Bluetooth, which can start and stop
data acquisition by sending commands to the BITalino device, see section 2.2.3. The sensors can be connected to the sensor kit through six analog inputs. However, the user is free to combine sensors and analog inputs as desired, with a maximum of six sensors connected simultaneously. When collecting data the sensors connected to each of the analog inputs are sampled at a given frequency, the sampled value is then transmitted to the micro-controller unit (MCU) as an electrical voltage (corresponding to the collected value). The MCU then converts each of the analog values to a digital value, which is combined to a data packet and sent to the connected Bluetooth device. These data packets are streamed to the connected device in real-time until the connected device sends a command to the BITalino to stop collecting data, or until the Bluetooth connection is lost.

**Energy usage**

The standard battery provided with the BITalino kit is a 550 mAh rechargeable LiPo battery. In the highest-demand scenario, that is if all the LEDs are on, and all the sensors are connected and collecting at 1000 samples per second, the BITalino uses around 65 mA per hour [19]; approximately 60 percentage used by the Bluetooth module, and 15 percentage by the LEDs. Using 65 mA per hour corresponds to a battery lifetime of 550 mAh/65 mA = 8.46 hours using the standard battery. This can be prolonged by not using all of the sensors or the LEDs, or by using a larger battery than the standard battery. By swapping out the standard battery with a 2-Ah battery the worst case lifetime can be prolonged to over 30 hours.

**2.2.3 Micro-controller unit**

In this section, the details of the micro-controller unit are described, based on its documentation found on the BITalino website [20]. The MCU is based on the MCU for Arduino, and is responsible for accurate and reliable data collection and real-time streaming of data over Bluetooth. The MCU can acquire and control up to six analog inputs, four digital inputs and four digital outputs at up to 1000 Hz. The analog inputs ports (A1-A6) are used to connect the interchangeable sensors to the BITalino device. When the device is streaming data, the sensors connected to the analog input ports are sampled by the MCU at a given frequency; this frequency is either 1, 10, 100 or 1000 Hz. Each of the sampled analog values are then converted to a digital value between 0 and 1023, represented by a ten bit integer. These digital values are then combined to a data packet and passed to the Bluetooth unit, as described in Table 2.3.

Out of the box the BITalino sensor kits are delivered with pre-installed firmware on the MCU. Based on this firmware the BITalino
device has three non-overlapping states (Idle, Live and Simulated). The BITalino sensor kits contains a LED-light, which blinks at different frequencies depending on the state. When the BITalino device is turned on the standby state (Idle) of the BITalino is entered. To indicate that the device is in this state the LED-light fades at 0.5 Hz (2 seconds between each illumination). In this state the MCU is waiting for commands from the connected Bluetooth device. To change the state from Idle to either Live or Simulated the commands listed in Table 2.1 are used. In the Live state the device is collecting and sending data sampled from the sensors to the connected Bluetooth device. The command that triggers the change to this state also specifies which of the six analog inputs the MCU should sample for values. Unless a different sampling frequency is specified (when in the Idle state), the chosen analog inputs are sampled at 1 Hz and sent to the connected device. To indicate that data is streamed in real-time the LED-light blinks at 1 Hz. Lastly, the third state (Simulated) streams synthetic generated data to the connected device. The synthetic data is available in form of sine waves, sawtooth waves and a pre-recorded ECG time series. To change between the three different states the commands in Table 3.1 can be used, each command is a combination of 8 bits; each entry in the table represent a bit of the command. The $A_1$ – $A_6$ fields are set to 1 to that the corresponding analog input should be sampled and 0 otherwise.

<table>
<thead>
<tr>
<th>Encoded command</th>
<th>Command explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>Change to Idle</td>
</tr>
<tr>
<td>1 0 $A_1$ $A_2$ $A_3$ $A_4$ $A_5$ $A_6$</td>
<td>Change to Live, with analog input selection</td>
</tr>
<tr>
<td>0 1 $A_1$ $A_2$ $A_3$ $A_4$ $A_5$ $A_6$</td>
<td>Change to Simulated, with analog input selection</td>
</tr>
</tbody>
</table>

Table 2.1: State commands

While it is possible to change to the Idle state from each of the other states, it is only possible to change to the Live and Simulated states from the Idle state. While in the Idle state, it is also possible to set a few other values, including setting the threshold for when it should be indicated that the battery is low, activating and deactivating digital outputs, and setting the sampling frequencies of the analog inputs. The commands for setting these values are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Encoded command</th>
<th>Command explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>6-bit threshold value</td>
</tr>
<tr>
<td>1 1 $D_1$ $D_2$ $D_3$ $D_4$</td>
<td>Set values of four digital output (D1-D4)</td>
</tr>
<tr>
<td>1 1</td>
<td>- - - - $F_s$</td>
</tr>
</tbody>
</table>

Table 2.2: Additional commands

### 2.2.4 Data packets

As stated in the previous section, in the Live and Simulated state the MCU puts the collected data into data packets. These data packets
are streamed in real-time to the connected Bluetooth device. Each of the
data packets consists of a 4-bit Cyclic Redundance Check code to
determine if the content of the data packet has been corrupted, a 4-bit
sequence number between 0 and 15, 1-bit values for each of the four
digital inputs (marked with D0-D3), and lastly the digital values of the
selected analog inputs (marked with A0-A5).

<table>
<thead>
<tr>
<th>A5</th>
<th>A5</th>
<th>A5</th>
<th>A5</th>
<th>A5</th>
<th>A4</th>
<th>A4</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4</td>
<td>A4</td>
<td>A4</td>
<td>A4</td>
<td>A3</td>
<td>A3</td>
<td>A3</td>
<td>A3</td>
</tr>
<tr>
<td>A3</td>
<td>A3</td>
<td>A3</td>
<td>A3</td>
<td>A3</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
<td>A2</td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
</tr>
<tr>
<td>A0</td>
<td>A0</td>
<td>A0</td>
<td>A0</td>
<td>A0</td>
<td>A0</td>
<td>A0</td>
<td>A0</td>
</tr>
<tr>
<td>CRC</td>
<td>CRC</td>
<td>CRC</td>
<td>CRC</td>
<td>CRC</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

Table 2.3: BITalino data packet

The size of the data packets depends how many of the analog inputs
were chosen when the collection was started; the data collection is
started by changing to either the Live or Simulated state. Each of the
analog inputs is represented by a 10-bit integer (between 0 and 1023) if
four channels are chosen when starting the data collection. If more than
4 analog inputs are chosen, the fifth and possibly sixth analog input are
represented by a 6-bit integer (between 0 and 63). Based on this, the
data packet is minimum 3 bytes and maximum 8 bytes. The payload \( B \)
of the data packet in bytes can be calculated by Equation 2.1, where \( b \)
is the number of bits of the payload (calculated by Equation 2.2) and \( N \)
is the number of channels. Equation 2.1 rounds the number of bits, \( b \), up
to the closest byte.

\[
B = \left( b + \left( 8 - 1 \right) \right) / 8 \tag{2.1}
\]

\[
b = \begin{cases} 
12 + 10 \times N & N \leq 4, \\
52 + \left( N - 4 \right) \times 6 & N > 4
\end{cases} \tag{2.2}
\]

If the device is collecting data from all sensors, analog channels A5 and
A6 get 6 bit resolution. But if only 5 channels are used, only the one
with the highest analog input number get 6 bit resolution. Moreover, if
only 4 inputs are selected all of the channels get 10 bit resolution, even
if A5 and A6 are among the selected channels.

2.2.5 Sensors and interpreting the data

A wide range of physiological, and physical, sensors are offered by
the BITalino team: light, temperature, accelerometer, EDA, ECG,
EMG, pulse sensor, electroencephalography (EEG), oxygen saturation
and respiration. Additionally, it is compatible with any third party
sensors provided that they have the same power and analog output
specifications as the BITalino sensors. The value sampled from the
connected sensors are, as mentioned earlier, either represented as 10-bit or 6-bit integers regardless of which sensors it originated from (and what unit it is measured in). To come around this, transform functions are specified in the documentation for each sensor. The transform functions scale the sampled value back to its original unit, as for instance the accelerometer values is scaled to a g-force value between -3 and 3.

2.2.6 Java SDK

The BITalino team offers software development kits (SDK) for multiple programming languages to ease the development of programs and applications using the BITalino sensor kit. The SDKs serve as an abstraction on top of the low-level interface used to communicate with the BITalino device, see Section 2.2.3. However, the SDKs do not provide methods for establishing a Bluetooth connection with the BITalino device since this is dependant on the operating system. They offer SDKs for many of the best known programming languages, such as Java, C++, Python, C# and MatLab to mention a few. In this section the Java SDK is explained, since this is used for developing Android applications. The Java SDK, like many of the other SDKs, are created by the BITalino community. Moreover, the Java SDK is created and maintained by Paolo Pires. The SDK mainly offers two abstractions, a BITalinoDevice and a BITalinoFrame. The BITalinoDevice is an object which manages the connection with the physical BITalino device. It exposes methods for starting, stopping and reading sampled values from the selected analog inputs at the selected sampling frequency. The read values are stored in BITalinoFrame objects, which hold all information contained in the data packet received from the BITalino device, see Table 2.3. In addition to these two abstractions the SDK offers implementation of the transform functions mentioned in Section 2.2.5.

BITalinoDevice

The BITalinoDevice is an object that holds information about the connection with the physical BITalino device, such as the sampling rate and analog channels that should be used when sampling values. Both the sampling rate and an int-array containing the selected analog channels are passed as arguments to the constructor when creating a new BITalinoDevice object. The sampling rate can either be 1, 10, 100 or 1000, while the list of analog channels contains the numbers of the selected channels. For example, to collect data from channel one and two, the analogChannel array passed to the constructor would contain the numbers one and two (analogChannel = [1, 2]). When creating a new BITalinoDevice the sampling rate and selected analog channels are saved, additionally the size of each data packet received from the BITalino device is calculated based on Equation 2.1 and 2.2.
public class BITalinoDevice{
    public BITalinoDevice(final int samplingate, final int[]
        analogChannels);
    public void open(final InputStream is, final
        OutputStream os);
    public void start();
    public void stop();
    public BITalinoFrame[] read(final int numberOfSamples);
}

Listing 2.1: Overview of the BITalinoDevice class

At this point a BITalinoDevice, which knows the sampling rate and
selected channels, is created. However, it is not yet specified where
to write and read messages to and from the physical BITalino device;
this is specified with a call to the open() method. This method takes
as arguments an InputStream and OutputStream to the connected
Bluetooth devices. These streams are later used to communicate with
the BITalino, by reading or writing values to them. As mentioned in
Section 2.2.3, the initial state of the BITalino device is Idle, in this
state the BITalino can receive commands for specifying the sampling
rate. The open() method saves the streams and writes a command
(see Table 2.2) to the output stream specifying the sampling rate for the
BITalino device. The start() method tells the BITalino device to start
collecting data from the selected analog channels by changing its state
from Idle to Live with the command specified in Table 2.1. While in
the Live state the BITalino device samples the connected sensors at the
selected analog channels at the selected sampling rate. These values
are sent from the BITalino device to the connected computing device
over Bluetooth, and are accessible on the computing device by reading
from the InputStream. To read the values from the input-stream the
read()-method is used. This method reads a number of data packets
(specified by the argument numberOfSamples) from the input-stream,
the size of each data packet depends on the number of selected channels
and was calculated when the BITalinoDevice object was created. The
content of each read data packet is extracted (based on the data format
in Table 2.3) and saved in BITalinoFrame objects. When extracting
the content of the data packets the check sum and sequence number
are controlled to check if the content is corrupted or the packet is out
of order. A list containing the BITalinoFrame of each data packet is
returned to the caller of the method when all of the data packets are
read. The BITalino device continues to collect data until it is told to stop.
This is done by calling the stop() method, which changes the state
of the BITalino device back to Idle, and closes the input- and output-
streams. In order to collect data from the BITalino device again, both
the open() and start() method needs to be called again.
BITalinoFrame

As mentioned, the content of each data packet received from the BITalino is stored as a BITalinoFrame object. It holds all the information of a data packet: the sampled analog values, the sampled digital values, the sequence number and the check sum. The stored information can be accessed through various get-methods, see Listing 2.2. In addition to these get-methods the BITalinoFrame object exposes methods for changing the values of the object, and for comparing BITalinoFrames with each other.

```java
public class BITalinoFrame {
    public int getCRC();
    public int getSequence();
    public int getAnalog(final int pos);
    public int getDigital(final int pos);
}
```

Listing 2.2: Overview of the BITalinoFrame class

The `getAnalog(pos)` method returns the sampled value of analog channel `pos`. These values, as mentioned earlier, are not necessary represented in the desired unit or range of values. The SDK offers various transform functions for scaling the sampled values to another unit and range.

Transfer functions

The SDK offers transform functions for most of the sensors supported by the BITalino device, see Listing 2.3. The scaling methods takes both the value (raw) and the analog channel it was sampled from (port) as arguments. This is done to compensate for the 6-bit resolution of the fifth and sixth analog channels, compared to the 10-bit resolution of the first four channels.

```java
public class SensorDataConverter {
    public static double scaleEMG(final int port, final int raw);
    public static double scaleECG(final int port, final int raw);
    public static double scaleAccelerometer(final int port, final int raw);
    public static double scaleAccelerometerWithPrecision(final int port, final int raw, final int min, final int max);
    public static double scaleEDA(final int port, final int raw);
    public static double scaleLuminosity(final int port, final int raw);
}
```
public static double scaleTMP(final int port, final int raw, boolean celsius);
public static double scalePZT(final int port, final int raw);
public static double scaleEEG(final int port, final int raw);
private static final double getResolution(final int port);
}

Listing 2.3: Overview of the SensorDataConverter class

All of the transform functions are based on the documentation for each of the sensors, found on the BITalino web page [4]. If for example luminosity is measured by analog channel number one, the scaled value can be obtained by combining the getAnalog() method and the transform function for luminosity, as shown in Listing 2.4.

double scaledValue =
SensorDataConverter.scaleLuminosity(1,
frame.getAnalog(0));

Listing 2.4: Sample code for scaling BITalino data

Where frame is a BITalinoFrame read from the input-stream of the connected Bluetooth device.
Chapter 3

Android OS

Android is an operating system developed by Google targeted mainly towards touch-based devices like smart phones and tablets. This chapter is based on the information found at the Android webpage [21], and describes the structure of Android and the main aspects of application development for Android.

3.1 Android architecture

The Android architecture [22] is structured as a software stack, with a Linux kernel at the bottom. Each layer in the stack adds an abstraction layer of the underlying hardware to simplify the task of application development. One of the abstraction layers introduces the Dalvik Virtual Machine, which is a virtual machine designed and optimized for Android; once an application is installed on a device it is running in its own virtual machine. The functionalities supported by the Android operating system are exposed to the application developer by the application framework, which includes the following key services:

- Activity Manager: Controls all aspects of the applications life cycle and Activity stack.
- Content Providers: Allow applications to share data with other applications.
- Resource Manager: Provides the applications with resources such as strings, color settings and, user interface layouts.
- Notifications Manager: Allows the application to display notifications and alerts to the user.
- View System: A set of views used to create application user interfaces. The application user interfaces are defined by combining different views.

At the top level of the software stack the Android applications are located, both the pre-installed ones and those that are installed by the user.
3.2 Application Components

This section introduces Activities [23], Services [24], and BroadcastReceivers [25], which are three of the core building blocks of an Android application. Activities are responsible for the user interaction, services perform (long-running) background tasks, and broadcast receivers receive broadcasted messages from other application components. These components are loosely coupled by the application manifest file AndroidManifest.xml, which describes the permission of the application as well as each component of the application and how they interact. All application components are started using an Intent, which will be further described in the section about interprocess communication.

3.2.1 Activities

An application can consist of multiple Activities, where each of the activities represent a single user interface. This is done by inflating Android layout files which define how the user interface looks. The activity implements the interaction logic of the user interface. An email application could for instance have one activity that shows a list of new emails, and another activity to compose an email. If an application has more than one activity, one of them is marked as the main activity which is presented when the application is launched. All implementations of an activity extend the Activity class and are therefore a part of the activity lifecycle.

Activity lifecycle

Activities in the system are managed as an activity stack. When a new activity is started it is placed on the top of the stack and becomes the running activity. The previous activity always remains below it in the stack, and will not come to the foreground of the stack again until the new activity exits. Each Activity is given one of four states:

- Running: A running activity is the activity on the top of the stack; the activity of the application the user currently uses. For instance, if the user is reading an email the running activity is the activity that shows the user the email.

- Paused: If an activity is not running but is still visible. This can for instance happen if a new activity that does not cover the whole screen is running, and is above it on the stack. A paused activity maintains all state and member information.

- Stopped: If an activity is no longer visible for the user. The activity is considered stopped if for instance the currently running activity covers all of the user interface of the activity in question. All information about the stopped activity is maintained, but since
the activity is no longer visible for the user it will often get killed by the system to reclaim resources.

- Killed: If an activity is paused or stopped it can be killed by the system to reclaim resources.

There are three key loops of interest within the lifecycle of an activity (see Figure 3.1) Firstly, the entire lifetime of the activity is between the first call to `onCreate()` and the final call to `onDestroy()`. Secondly, the visible lifetime of the activity, corresponding to the visible state (i.e. when the application is visible to the user), is between the `onStart()` and `onStop()` calls. Lastly, the foreground lifetime of the activity, corresponding to the running state (i.e. when the user got the activity in focus). This is between the `onResume()` and `onPause()` calls, as illustrated in Figure 3.1. The difference between the second and third loop is that the not all visible activities are in focus, i.e. currently being used by the user.
3.2.2 Services

A Service is an application component that runs in the background to perform long-running operations. For example, a service might play music in the background while the user is in a different application, or might collect data from a sensor without blocking user interaction with an activity. Application components can start a service and it will continue to run in the background even if the user switches to another application. An Activity on the other hand will not continue to run if the user switches to another application, because this will trigger a call to the Activity's onPause() method. There are two types of services, separated by the way they are invoked; a Service can be either started or bound.

A started service is typically used to perform a single operation, such as downloading a file, and is started by a call to startService() by another application component. Once a service is started, it runs in the background even if the application component (such as an activity or a broadcast receiver) that started it is killed.

A bound service is used if interaction between the service and the application component that started the service is desired. A bound service starts when an application component binds to it by calling bindService(). Multiple application components can bind to the same service at a time, and the service runs as long as at least one application component is bound to it. A bound service offers an interface for interaction with the service, such as sending request or retrieving results.

3.2.3 Broadcast Receivers

A BroadcastReceiver is an application component that receives broadcasted messages (called Intents) from other application components. For instance, a Service which downloads a file would typically broadcast a message, i.e. Intent, to inform other application components that the download is complete. A broadcast receiver subscribes to certain message, i.e. Intent, and when these messages are broadcasted the BroadcastReceiver receives and processes them. For instance, a BroadcastReceiver alerts the user with a notification, that tells the user that the download is finished, when an Intent is received.

3.3 Processes and threads

In most cases, all application components of an application run within the same process [27]. This process is created when any of the code of the application needs to be executed, and will remain running until it is no longer needed, or the system needs to reclaim resources for other applications.
3.3.1 Process’s lifecycle

The lifetime of an application’s process is not directly controlled by the application itself. The system determines which of the application processes should run, and can kill a process when desired. If the system is low on resources, such as memory, the system can choose to kill an application. To determine which application processes to kill Android gives the various processes different ranks (priority). The rank of a process is determined based on how important the process is for the user experience. A process is given one of five ranks, based on which requirements are met:

1. A foreground process is a process containing an application component that is very important for the user experience. A process is considered a foreground process if the process contains a running activity, or another application component running in the foreground.

2. A visible process is a process containing a visible activity, as explained in Section 3.2.1.

3. A service process is a process containing a started service, playing music for instance.

4. A background process is a process containing a stopped activity, as explained in Section 3.2.1.

5. An empty process is a process that is not holding any active application component. These processes are only kept as a cache to improve start up time the next time the application is launched.

A process is given the highest rank found among all of the application components of the process. Additionally, the rank of a process might be increased because other processes depend on it. The rank of an application component can also be increased by starting it in the foreground using startForeground(), giving the application component (and its process) the rank of a foreground process.

3.3.2 Threads

Upon application launch the system creates a thread for executing the application, also known as the main thread [27]. This thread is responsible for dispatching events to the user interface. All components that run in the same process are instantiated from the main thread, and system calls to each component are dispatched from the main thread. The system does not create a new thread for each application component; all executions are performed on the main thread unless the developer specifies it to run on another thread. If the application performs intensive computations in response to a user interaction, the performance of the device can be poor. For instance long-running operations, such as networking, will block the main thread and no events can be dispatched.
From the user's perspective the application appears to hang.

Additionally, the tool kit used to updated the user interface (UI) of the application is not thread safe and convention is to only update the user interface from the main thread; also called UI thread. With a single thread responsible for dispatching application components and updating the user interface it is important to not perform long-running or blocking operations on the main thread. The developer should rather run long-running operations on a separate thread. This can either be done by using an `AsyncTask` or simply spawning a new thread.

### 3.4 Interprocess communication

As earlier mentioned, each Android application runs within its own virtual machine. Therefore, many traditional interprocess communication (IPC) mechanisms that rely on a shared memory region, like mailboxes and pipes, are useless. Instead, Android offers communication between processes by performing remote procedure calls [27]; a method is called locally in a process, but executed remotely in another process. The method call is decomposed into primitives the operating system can understand, then transmitted to the remote process by using a kernel buffer, in the remote process it is recomposed and executed, lastly the return values are transmitted in the opposite direction. This is made possible by the Binder framework [28] [29], which is a part of Android's Linux kernel. To perform IPC the application needs to bind to a remote `Service`, using `bindService()` as described earlier. When bound to the remote service a proxy is returned, this proxy is used for communication with the remote service. In the remote service, a thread pool exists for executing requests made to the proxy. In Figure 3.2, process A has bound to process B, which got multiple threads ready for execution. The proxy decomposes the method calls made to it, these decomposed method calls are transferred to the remote process by the Binder driver in the Linux Kernel, and are executed by one of the threads of the thread pool in the remote process.
Figure 3.2: Illustration of Binder communication [29]

The code to define an interface between processes is tedious to write. Luckily, Android offers a language for defining it; Android Interface Definition Language [30].

Another IPC mechanism offered by the application framework is to pass messages using Intents [31]. An Intent is a message object used to request an action from another application component. Intents can either be addressed directly by specifying the application component (explicit Intent), or by specifying the requested action (implicit Intent) instead of the component to perform it. Intents are used to request activities and services to start (explicit), and to deliver a broadcast (implicit). To receive implicit intents, the interested application component needs to register a BroadcastReceiver listening for intents with the desired action by declaring an intent-filter.

3.5 Files

The file system [32] used by Android is similar to the file system on other platforms. It consists of two storage areas, internal and external storage. Some devices divide the internal phone storage into internal and external partitions, such that there are two storage spaces even without a removable storage medium.

3.5.1 Internal storage

The internal storage is always available. When the application is saving files to the internal storage the files are only accessible by the application that created the file. When an application is uninstalled, the files it saved to the internal storage are deleted. No permissions are needed to read and write files in the internal storage, since the internal storage of the application is intended to only be used by the application itself.
### 3.5.2 External storage

The external storage might not always be available, since the external storage might be removable. Files located in the external storage are readable by other applications. The external storage is typically used for files that do not require access restrictions and/or files shared with other applications. The files in the external storage are accessible by a file explorer when the mobile device is connected to a computer. Writing to the external storage requires special permissions. To be able to modify files in the external storage the application needs to request the permission to write to the external storage in the manifest file, see Listing 3.1.

```xml
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Listing 3.1: Request of permission to write to the external storage

### 3.5.3 File size

The file size is limited by the file system, as for all other operating systems. The file system might be different on the internal and external memory of a device. Actually, most SD card uses FAT32 limiting the file size on the SD card to 4GB. The files system used to store files on Android devices internal memory varies from device to device. However, the majority of recent Android devices uses ext4 [33], this file system does not limit the file size to 4GB as FAT32 does.

### 3.6 Energy saving

Most Android devices and other mobile devices, have a limited amount of energy available. To prolong battery time, Android uses various energy saving mechanisms [34]. Android devices first dim the screen, before they turn off the screen. Some time after the screen is turned off, the frequency of the CPU is turned down to further reduce the energy consumption. Some applications might want to keep the CPU running at a higher frequency even when the screen is off, and `WakeLocks` enable this. A `WakeLock` can be used to keep the CPU running, and prevent the screen from dimming or turning off. Until Android 6.0 it was possible to force the CPU to be ON by using a `WakeLock`, which allow the application to run even while the screen is turned off.

In Android 6.0, also know as marshmallow, Doze [35] was introduced. To save power Doze puts applications into deep sleep, overriding `WakeLocks`, when the device is not in use. Three requirements need to be fulfilled to enter Doze mode.
1. The device needs to be unplugged.
2. The device needs to be stationary.
3. The screen needs to be turned off.

While in Doze, the system tries to save energy by restricting applications network access and CPU-intensive services. The applications are prevented from using the network and the jobs, syncs and standard alarms of the applications are postponed until the device wake up again.

![Figure 3.3: Illustration of the effects of Doze [35]](image)

Periodically the device wakes up, the up to sync data and perform the postponed activities, before it enters Doze again. The time between these maintenance windows increases over time, but as soon as the device is moved, screen is turned on, or connected to a charger the system exit Doze.
Part II

Design and implementation
Chapter 4

Analysis and High-level design

4.1 Requirement Analysis

In this section the requirements for the data acquisition system are analysed. These requirements build on the challenges and premises identified in the introduction. To summarize the findings in the introduction:

1. The data acquisition should be decoupled from the data management and data analysis, making it possible to change the data source to acquire data from other sensors (data sources) without changing the data management and data analysis part of the software.

2. The data acquisition should hide the details of the data collection from the developer of the application which uses the collected data. Developing these applications should not require sensor specific technical knowledge.

3. It should be easily extensible with respect to new data sources.

4. The implementation to support a data source should be reusable across applications. To reduce the duplicate work and code.

5. The data should be offered to the applications in a way such that the context of the data acquisition can be determined from the provided data itself.

Based on these findings both functional and non-functional requirements are found. The most basic functional requirement is that the data acquisition system obviously needs to be able to collect data. Without this functionality the system would not be a data acquisition system. To ensure an acceptable quality of the collected data the system needs to be resilient. In order to be resilient multiple functional requirements are identified. Firstly, none of the collected data should be lost in the
system. Secondly, the data acquisition system needs to be stable, ensuring that the data acquisition does not crash. Lastly, security measures are needed to offer privacy of the collected data. Offering privacy of the collected data is important, especially for medical data. However, this will not be covered in this thesis, only briefly discussed in the Future work section. Furthermore, the data acquisition needs to use

When abstracting and decoupling the data acquisition away from the application developer a well defined way of managing (starting and stopping) the data collection is necessary. This yields the need of offering an interface to the data acquisition system, as well as a standardized way of representing the data. Furthermore, the context of the data collection (i.e. what the sensors measures) should be deducible from the data representation. Hence, the representation should contain information about the capabilities of the data acquisition (i.e. what the sensor measure) and the unit it is measured in (metric). For example, if a sensor to measure oxygen saturation is used for data collection, than the data provided by the data acquisition system should include the metadata that tells that oxygen saturation is measured and in which metric.

The motivation of creating the data acquisition system is that the implementation of supported sensors can be reused across applications. The system should be able to support both currently available data sources, and the ones available in the future. To achieve this the data acquisition system needs to be extensible and open, enabling anyone to add support for the data source they desire (if not supported already). In order to successfully do this, the system should be as easy as possible to extend and maintain the system. Additionally, the resource usage of the system should be moderate. For instance, the CPU-usage should be kept low for all possible configurations of the sensor wrappers (i.e. for all possible sampling frequencies) to enable other applications to use the collected data. If the data acquisition system used all of resources of the mobile device, no other application could use the collected data. Therefore, the resource usage of the data acquisition should be kept low.

To recap the requirements presented in this section, the system should:

1. be able to collect data,
2. offer an interface for managing the data acquisition,
3. offer the data in a descriptive and well defined way,
4. be easily extensible,
5. be resilient in terms of
   • data loss,
   • stability,
   • security,
• low resource usage

Next, the high-level design of the extensible data acquisition system, supporting the identified requirements, is discussed.

4.2 High-level design

To meet the requirements presented in the previous section a top down approach is used. There is two fundamentally different ways of creating a data acquisition system. The system could either be push-based or pull-based. In a pull-based data acquisition system the application that used the collected data needs to request all of the data it receives, while in a push-based system the data is presented to the application as soon as it is available. Unlike in a pull-based architecture, the receiving applications does not have to request data in order to receive it in a push-based architecture; it is sent to the receiving application as soon as it is collected. Therefore, the data management is less complex than in a pull-based architecture, where the receiving application can choose what data it is interested in. Additionally, in this data acquisition system all of the collected data is of interest of the receiving application. We therefore want to design a push based data acquisition system that gathers data from sensors that are connected to a smartphone and uses the collected data.

The most naive solution to achieve this would be to create a single application which supports all existing and future data sources, as shown in Figure 4.1.

This solution would be fine as long as all possible data sources are covered by the implementation and no extension is needed. Extending an application like this is difficult since the new extension would have to be compiled together with the existing source code. Hence the integration of a new data source can only be done by a developer who got access to the source code, and therefore the system is not easily extensible by any developer.

To make the system more extensible the core of the application should be kept unchanged when adding support for a new data source. To further build on the concept of keeping the core of the application unchanged, two high-level design alternatives are proposed below.

4.2.1 Protocol layer solution

Different data sources might use different data exchange protocols to communicate. By adding a protocol layer on top of the data source specific protocol the difference is removed and all data sources use the same protocol to communicate. Additionally, a protocol layer needs to be added at phone as well to enable communication between the entities. This follows the concept of keeping the core of the application
Moving the support of a new data source from the system to the data source introduces some constraints on the data sources. The data sources must for instance be programmable, which excludes certain data sources from the system, such as simple sensors without a microcontroller unit. Moreover, adding a protocol layer to the programmable sensors, e.g. BITalino, requires low-level technical skills and is probably much harder than adapting the application to the protocol. Adding a protocol layer on top of the data source solves the issue that different data sources might use different data exchange protocols, but it does not solve the issue that different data sources might use different link layer technologies. This still needs to be handled by the protocol layer on mobile device. This introduces the same problem as for the naive solution; to extend the system to support a new link layer technology the source code of the whole application is required. Additionally, using the same protocol for all of the data sources limit the functionally of the data source to the functionality supported by the protocol.

Making all the data sources behave the same does not meet all of the requirements. It only solves one of the two problems with the naive solution. Another possible solution is to separate the system into different components with different responsibilities.
4.2.2 Modular solution

We desire the core part of the application to be the same regardless of the Link Layer technology and communication protocol used by the data source. To achieve this modularization a separation of the software is proposed. We propose to separate the software into two different components, a provider component and a sensor wrapper component, to achieve reuse of the functionality that is shared among the different data sources. The provider component is responsible for the functionality which is common for all data sources (such
as starting and stopping the acquisition), while the sensor wrapper component is responsible for the data source specific functionality (such as communicating with the data source). A sensor wrapper is tailored to fit the Link Layer technology and data exchange protocol of one particular data source.

Figure 4.3: Architecture of a software separating solution

The sensor wrapper applications are responsible for connectivity and communication with the data source. They collect data from the data source and push it to the provider application using interprocess communication. The provider application is responsible for managing the
sensor wrappers (i.e. starting and stopping data acquisition) and for processing the data it receive from the sensor wrapper applications. Everything that is independent of the data source should be a part of the provider application. The software separation enables the possibility to reuse the sensor wrapper applications for different provider applications.

This proposition is extensible to all data sources as long as the Link Layer technology is supported by Android, and it does not limit the data sources to sensors; even files, databases and software probes could in theory be used as data sources, for instance to simulate the acquisition of a previously acquired dataset. To extend the system with support for a new data source a sensor wrapper application for the data source is required. The fact that the whole system consists of applications makes it easily available to the user; the user can simply download the desired sensor wrapper applications to extend the system provided that the sensor wrapper application is available. For instance, if a user wants to collect data using a BITalino sensor board, and a sensor wrapper application for the BITalino sensor board is available, the user can download the BITalino sensor wrapper application to the mobile unit and use it.

There are some downsides with the separation into different components. The separation introduces an overhead due to interprocess communication which might be costly and increase the complexity of the code. Regardless of the increased complexity and the overhead introduced by interprocess communication, the flexibility and extensibility gained by separating the data source specific functionality from the functionality common for all data sources makes it the design best suited to meet the requirements.

4.3 Data

The chosen high-level design involves sending a data stream with potentially high tuple frequency between different components of the system. Therefore, an overview of what the data packets look like is presented before going into the details of the two software components. In this chapter certain requirements are given to the data, and different data serialization formats are compared. Lastly, data packets which meets the requirements are proposed.

4.3.1 Requirements

As mentioned earlier, different sensor might sample data in different contexts, and some applications are strictly dependant on knowing the context of the data acquisition; it is critical to know what the data values measure. Some data sources (as the BITalino sensor kit) might consists
of multiple inputs (hereafter referred to as data channels), for each of which the context of the data acquisition might be different. When exposing all sensors and data channels through one common interface this context needs to be specified as metadata. Therefore, it should be possible to determine the context from the data alone. To summarize, from the data it should be possible to:

- distinguish which sensor wrapper and data channel the data originated from,
- determine the capabilities of the sensor that sampled the data (i.e. EEG, ECG, LUX),
- determine which unit it is measured in (i.e. Celsius or Fahrenheit for temperature),
- give a description of the data channel (for instance give information about the placement of the sensor),
- determine when the data was sampled (with a timestamp).

### 4.3.2 Data format

There are many applications revolving around sending and receiving data over the internet. The increase in data exchange over the internet has made the selection of a proper data serialization format increasingly important. Data serialization is the process of writing the state of an object to a stream and of rebuilding the stream back into an object. The two most common modern data serialization formats are XML (eXtensible Markup Language) [36] and JSON (JavaScript Object Notation) [37]. They are both widely used and well documented.

XML is a subset of the Standard Generalized Markup Language (SGML) that was developed by members of the W3C [38]. The fundamental design considerations of XML include simplicity and human readability. Amongst the design goals of XML, the W3C specifies that "XML shall be straightforwardly usable over the Internet" and "XML documents should be human-legible and reasonably clear."

A XML-document is a format for representing data, see Listing 4.1. The format mainly consists of markup and content, differentiated by syntactical rules. The most common markup is called a tag, and starts with `<` and ends with `>`. The content is placed between two of these tags. Namely, after the start-tag (as `<first>`) and before a stop-tag (as `</first>`). This combination of a content and tags is called an element. A XML structure consists of combinations (often nested) of such elements. It has a large user base, and is used widely in various web services. Despite its widespread use, XML is often criticized for being overly verbose.
Another common data serialization format is JSON, which is designed to be a data exchange language which is human readable and easy for computers to parse and use. JSON was developed by Douglas Crockford who used it for several years before launching www.json.org. Soon afterwards both Yahoo and Google began to offer web services in JSON. As XML, it is a format for encoding data. However, instead of using tags to mark the content of the data, JSON consists of key-value pairs (as shown in Listing 4.2). A JSON representations can consists of multiple key-value pairs (as "firstname":"John"), and of arrays containing key-value pairs. JSON is often praised as a lightweight and more efficient alternative to XML, since it does not have as much markup overhead.

```json
{
    "firstname":"John",
    "lastname":"Smith"
}
```

While JSON and XML are the most widely used data serialization formats, binary formats are quickly gaining momentum. Because both JSON and XML are text-based, they will need to be parsed character by character, thus imposing a limit on de-serialization speed. The binary formats are not subject to the common disadvantage of the JSON and XML formats.

**Comparing alternatives**

In this section, a number of papers that investigate and compare data serialization are presented and summarized. At the end of this section a choice of data format for the system will be done based on the content in the papers.

Nurseitov et al. [39] compared XML and JSON formats for a client-server environment. They showed that JSON is faster and uses fewer resources than XML for this environment. Eriksson and Hallberg compared JSON with YAML (Yet Another Markup Language) [40]. They
showed that YAML is more functional and readable than JSON, but JSON is much faster than YAML for both serialization and deserialization. This fact, combined with the lack of web services using YAML, disqualifies YAML for being a recommended format for data serialization on mobile phones. Sumaray and Makki [41] compared the efficiency of XML, JSON, and two binary serialization formats, Thrift and ProtoBuf on a mobile platform. The focus of the comparison is the data size, serialization speed and ease of use on a mobile platform. They show that the XML data format is largely inferior to the other serialization formats with respect to serialized size and speed. JSON on the other hand avoids the disadvantages of XML while maintaining many of the advantages. It has a large user base, is human readable, and is used in a large number of web services. The binary formats were found to be faster and smaller than the text-based formats, but they are not as adaptable since data is sent in a binary format.

From the various papers we can conclude that JSON is faster than XML, but a bit slower than Binary buffers. Even though JSON is a bit slower than Binary buffers it got multiple good libraries which makes it easier to use. The fact that JSON is human readable and easy to use, while still performing good, makes it well suited for the system. When passed between processes and over the network, a JSON-object is represented as a string without whitespaces.

4.3.3 Data packets

To meet the requirements presented in the start of this section all the desired information could be placed inside each JSON-object. However, this introduces overhead since information about data types and metrics are sent multiple times. So a better solution would be to send all data that does not change during the acquisition as metadata at the start of the acquisition. The data collected from the sensor on the other hand is not the same during the acquisition and hence it should be sent when a new value is available. This clear difference leads to two different data packets, namely metadata packets and data sample packets. Differentiated by a type key in the JSON objects representing the data packets. The type key is given the value meta and data for metadata packets and data packets respectively.

Metadata

Metadata packets are sent at the start of the acquisition to describe the context of the data collected from the different sensor wrappers. The data that remain constant and could be sent as metadata are the name of the sensor wrapper and information about the context of each of the data channels. The information about each of the data channels includes the capability, metric and description of the data acquisition for each data channel. The system could potentially acquire data from
multiple sensor wrapper applications and data channels at the same
time, so each sensor wrapper application is given a sensor wrapper ID
and each channel is given a channel ID. The combination of sensor
wrapper ID and channel ID uniquely identifies each channel in the
system. Provided the necessary metadata it is possible to keep a table
of the information about each combination of sensor wrapper ID and
channel ID as shown in Table 4.1.

<table>
<thead>
<tr>
<th>Wrapper ID</th>
<th>Channel ID</th>
<th>Wrapper name</th>
<th>Data type</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>BITalino</td>
<td>EEG</td>
<td>Microvolt</td>
<td>Head, left</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>BITalino</td>
<td>EEG</td>
<td>Microvolt</td>
<td>Head, right</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Database</td>
<td>LUX</td>
<td>Percent</td>
<td>Window</td>
</tr>
</tbody>
</table>

Table 4.1: Illustration of metadata

Since JSON is the chosen data format for the data packets, the
metadata packet is represented in the following JSON object. The
name of the sensor wrapper is marked with “name”, and the information
about each data channel are given as elements in a JSON array.

```json
1 {  
2     "type": "meta",  
3     "name": "InPhoneSensor",  
4     "id": 0,  
5     "channels": [  
6         {  
7             "id": 0,  
8             "type": "ACC",  
9             "metric": "G",  
10            "description": "Phone, X"  
11         },  
12         {  
13             "id": 1,  
14             "type": "ACC",  
15             "metric": "G",  
16            "description": "Phone, Y"  
17         },  
18         {  
19             "id": 2,  
20             "type": "ACC",  
21             "metric": "G",  
22            "description": "Phone, Z"  
23         }  
24     ]  
25 }
```

Listing 4.3: A JSON structure describing the encoding of the metadata

In the example in Figure 4.4 a metadata packet from a sensor wrapper
with three data channels is shown. The metadata states that the sensor
wrapper measures accelerometer values in the x-, y-, and z-direction in a phone.

Data reading

To link the sampled values to the metadata the data sample packets needs to contain the same combination of sensor wrapper ID and channel ID as the metadata sent at the start of the acquisition. A timestamp slot also needs to be provided in the data sample packet to meet the requirement that the data is timestamped. The timestamp is marked with time in the JSON object. If multiple values from the same sensor wrapper have the same timestamp (like it possibly could for BITalino) they can be combined into the same data packet as shown in Figure 4.5.

```json
{
  "type": "data",
  "id": 0,
  "time": "13:28:59:365",
  "data": [
    {
      "id": 0,
      "value": 0.057708740234375
    },
    {
      "id": 1,
      "value": -0.0457763671875
    },
    {
      "id": 2,
      "value": 9.865188598632812
    }
  ]
}
```

Listing 4.4: A JSON structure describing the encoding of a data reading

In the example in Figure 4.5 a data packet from sensor wrapper 0 with values for channels 0, 1 and 2 is shown. From the combination of sensor wrapper ID and channel ID we can use the metadata to get more information about the different data channels. From the metadata in Figure 4.4 we know that the combination of sensor wrapper ID 0 and channel ID 0 measures accelerometer values in a phone.

4.4 Separation of concern

In this section, the concerns the system has to address are listed, and the corresponding part of the system that is responsible for
solving it is specified. From the high-level design we know that adapting to different sensors is the responsibility of the sensor wrapper applications, while the functionally that is common between data sources is the responsibility of the provider application.

4.4.1 Linked layer

To communicate with a data source a connection is needed. Different data sources could possibly use different link layer technology, so the system needs to be extendible with respect to the Link Layer. The Link Layer technologies used by different data sources are numerous. For example is Bluetooth used by some sensors, like BITalino, while ANT+ is used by others, like Shimmer. The Link Layer technology is determined on the data source, hence it is within the scope of the sensor wrapper application.

4.4.2 Data exchange protocol

There is no standard data exchange protocol used by all data sources, each data source could possibly use an unique one. The data exchange protocol is used for communication with the data source, which includes sending, receiving and extracting the content of the data packets it collects. As for the Link Layer technology, the data exchange protocol is determined by the data source and is therefore within the scope of the sensor wrapper.

4.4.3 Using the collected data

From the high-level design we know that the collected data is passed to the provider application through interprocess communication. How the data is used is independent of the data source, and it is therefore within the scope of the provider application. The provider application is responsible for using the data it receives as JSON-objects from all of the sensor wrapper applications.

4.4.4 Sensor wrapper management

In some scenarios, only a subset of the sensor wrapper applications that are available on the mobile phone are required or are useful. Therefore, user should be able to select which sensor wrapper applications to utilize. This is the responsibility of the provider application since this involves all sensor wrapper applications and not a specific one.

4.5 Provider design

The provider application is an entity that uses the sensor wrapper applications to collect data. As mentioned earlier, each data source supported by the system got a corresponding sensor wrapper application.
The provider application can use any of the sensor wrapper applications to collect data through one common interface. Once started, the sensor wrapper application pushes the collected data to the provider application through IPC. When the data is received, it is processed as desired by the developer of the provider application. In this section the management of the sensor wrapper applications is described, before the processing of the received data is discussed. Lastly, the possible states of the provider application, and the transitions between them, is presented.

### 4.5.1 Sensor wrapper management

From the high-level design we know that there is one sensor wrapper application for each data source, so multiple sensor wrapper applications can be available on one mobile unit. The provider application can use any of the installed sensor wrapper applications to collect data through one common interface. Therefore, it is naturally to expose functionality for starting and stopping data collection with the sensor wrapper applications through this interface. Depending on the situation the provider application might want to acquire data using either all the sensor wrapper applications, or, in some use cases, only a subset of them. To enable data acquisition where only a subset of the sensor wrapper applications is used, the provider application should be able to specify which of the sensor wrappers to start when starting the data acquisition. Additionally, it should be possible to stop a subset of the started sensor wrapper applications. To enable data acquisition with a subset of the sensor wrapper applications the provider application should be able to choose which one to use. This is done by assigning an ID to each of the sensor wrapper applications, which is used when starting the data acquisition to specify the subset to use; only the selected sensor wrapper applications should be notified to start when starting the data acquisition. The same ID is used to specify which of the started sensor wrappers to stop. The possibility to select which of the started sensor wrappers to stop enables the possibility to only stop a subset of them. Additionally, the sensor wrapper applications are shared between multiple provider applications, which all start and stop the data acquisition through the same interface. In order to achieve this, the sensor wrapper applications needs to know where to send the collected data. Therefore, the interface provides the possibility to specify where to send the collected data as a part of the start command.

### 4.5.2 Sensor wrapper discovery

However, the user is free to install and uninstall applications at any time. This means that the set of available sensor wrapper application on the mobile phone can change between the launches of the provider application, and a mechanism to determine the ID of the available sensor wrapper applications is required. As stated, the set
of available sensor wrapper applications might change. To cope with this a mechanism to check for available sensor wrapper applications is designed. One possible way to update the set of available sensor wrappers is to fetch the values from a file that contains the ids of all the available sensor wrapper applications. Another, more flexible and dynamic, solution is to let the available sensor wrapper applications register when the provider application wants to know the available sensor wrappers. This can be achieved by a handshake between the applications. The provider application sends out a message, which all the sensor wrapper applications listen for. The sensor wrapper applications respond to this message, letting the provider application know that it is available for data acquisition. The response contains an ID, which later can be to address the sensor wrapper (for instance to start or stop the data acquisition). By performing this handshake when launched, the provider application is aware of all the available data sources when it is running.

4.5.3 Data processing

The chosen high-level design involves interprocess communication between the two application types. A provider application is able to start data collection from one or multiple sensor wrapper applications concurrently. When the data acquisition is started, the selected sensor wrapper applications collects data and pushes it to the specified provider application as JSON strings. When the provider application receives the data it can in principle process the data as it desires. Whether means to store the received data to a file, perform real-time analysis on the received data, or forward the received to another application or computing device is up to the developer of the provider application. Some processing might be dependent on the context of the collected data to execute correctly. This could be to determine what a data values measures or what unit it is measured in. This information is sent as metadata (see Section 4.3.3), and is available for the provider application. In this thesis, we have chosen to implement a prototype of a provider application that either can store the received data to a file or send it from the computing device to another device using TCP. Since the high-level design involves sending messages from the sensor wrapper application to the provider application, the interface naturally needs to expose a message interface to the sensor wrapper applications for sending strings to the provider application. However, multiple sensor wrappers applications might send such messages simultaneously and rapidly. The provider applications needs to take this into account, such that message handling neither looses messages nor becomes a bottleneck. This could for instance be solved by handling the messages sequentially, or by implementing a multithreaded solution that handles multiple messages concurrently.
4.5.4 States

A provider application consists of two states, it can either be *Idle* or *Live*. In the *Idle* state the provider application has not started any sensor wrapper applications yet, and therefore it does not receive any messages in this state. The provider application is considered to be in the *Live* state if it has started data acquisition and it receives data from at least one sensor wrapper application. When the provider application is in the *Live* state it processes the data it receives from the started sensor wrapper applications.

The transition from the *Idle* state to the *Live* state is triggered when the provider application starts collecting data from at least one sensor wrapper application. Which in turn is initiated by a start command that is sent to the selected sensor wrappers telling them to start collecting and streaming data to the provider application. The transition from *Live* to *Idle* is triggered when all of the active sensor wrapper applications are stopped, by a stop command. Both the start and stop command specify which sensor wrapper applications it apply to, making it possible to start and stop data collection from a selected data source when desired. A provider application would typically initialize the sensor wrapper discovery handshake to know which sensor wrapper applications it can use for data collection, and use the IDs acquired from the handshake to start and stop acquisition.

4.6 Sensor wrapper design

A sensor wrapper is responsible for establishing a connection to and collecting data from exactly one specific data source, and for sending the collected data to the provider application which told the sensor wrapper to start. A data source (such as the BITalino) might have multiple sensors attached to it, referred to as data channels. However, only one sensor wrapper is required for each data source even though it consists of multiple data channels. Each sensor wrapper application is tailored to fit the Link Layer technology and communication protocol of one specific data source. When started by a provider application it collects data from that specific data source and pushes the collected data to the provider application as JSON-strings. In this section the responsibilities and the states of the sensor wrapper application are explained.

4.6.1 Adapting to the data source

Each sensor wrapper application is used to collect data from one specific data source. Unfortunately, different different sensors might use different Link Layer technologies and communication protocols. These differences are hidden from the provider applications by the common interface used to collect data from any of the sensor wrapper applications. A sample implementation of the adaption to the Link Layer technology
To be able to collect data from a data source the sensor wrapper application primarily needs to establish and keep a connection to the data source. Different data sources might use different Link Layer technologies (such as cable, ANT+, Bluetooth, WiFi and ZigBee), which might establish connections differently. For a data source that uses Bluetooth (such as BITalino) an initial pairing and code to establish a connection is required, while a cabled connection might not require any code to establish a connection between the entities. Once a connection with the data source is established an agreement of how to format the interchanged data is required. This format might vary between data sources, which often interchange data with minimalistic byte sequences. The interchanged data includes all data sent between the communication entities, such as the commands for starting and stopping the data acquisition and format of the collected data. BITalino, for example, uses various combinations of eight bits to represent the commands sent between the communication entities, see Table 2.1 and 2.2. The sensor wrapper application uses these commands to manage the data collecting from the data source. The developer of a sensor wrapper application therefore needs to know the data source specific details and commands to connect and communicate with the data source.

4.6.2 Configurations

Various data sources might provide the possibility to configure the data acquisition, e.g. sampling frequency for BITalino. The data source might be able to collect data at different sampling frequencies or it might be possible to turn on and off the sampling for certain sensors connected to the data sources. From Chapter 2, we know that the BITalino sensor kit can sample data at four different frequencies and that any combination of the six analog inputs can be chosen to collect data from. These configurations could have been exposed as a method in the common interface between the two application types. However, this could put too strict limitation on the possible configurations; some parameters of current or future sensors might not be possible to configure. Instead, a design where each sensor wrapper application exposes the possible configurations as a graphical user interface is chosen. This choice does not put any restrictions on the possibilities of the configurations.

4.6.3 Forwarding the collected data

A handshake mechanism is used to determine the available sensor wrapper applications. The handshake is initialized by a message sent out by the provider application to all of the sensor wrapper applications. Upon receiving such a message each sensor wrapper application
responds with a message containing an ID that later is used to start and stop data acquisition with this particular sensor wrapper. When the data acquisition is started by a provider application, the selected sensor wrapper applications connect and start data acquisition from their data source. The sensor wrapper application collects values from the data source (according to the configuration selected by the user), and forwards these values to the provider application. The protocol between the data source and the sensor wrapper is data source specific and the data packet used to represent the collected data might vary between data sources. The data collected from the BITalino sensor board is represented by 3 to 8 bytes (see Table 2.3), where each of the collected values are represented as 10- or 6-bit integers. These values are then extracted from the data packet and a JSON-object is created of the values, according to the data standardization specified in Section 4.3.3. The sensor wrapper application continues to forward the collected data values to the provider application until a stop command is received from it.

Most sensors have only small finite (or even no) data buffers for data samples. These buffers are often used to temporarily store the data packets before they are sent from the data source to the connected computing device. If the sensors samples and creates new data packets faster than sensor wrapper application consumes the data, data loss will occur. This is because the temporary buffer is filled up, resulting in data loss due to buffer overflow. To avoid this, anything done between requesting data packets from the data source (such as extracting values or creating JSON-objects) should always take less time than time needed to sample and dispatch a new data packet by the data source. Therefore, the computational work, such as extracting and creating JSON-objects of the data, should be done in another separate thread than the thread designated for communicating with the data source.

4.6.4 States

As for the provider application, the sensor wrapper application consists of two states, it can either be in the Idle or Live state. When the application is in the Idle state it is disconnected from its data source, and it is waiting for commands from a provider application. These commands can either be the handshake message sent from the provider application or a message telling the provider application to start data acquisition.

When a start message is received a transition to the Live state is triggered, the sensor wrapper application connects to the data source and starts collecting data. The collected data is converted to JSON-objects and continuously sent to the provider application specified by the start message. As long as the application is collecting and sending data it is consider to be in the Live state. It continues to collect data until a stop message is received from the provider application that told it to start. As expected, when a stop message is received the sensor
wrapper application stops collecting data and disconnects from the data source.

4.7 Combing provider and sensor wrapper applications

The data acquisition system consists of two abstractions. Sensor wrappers, which enables any application to collect data from a data source, without having to know the technical details of the data source, through one common interface. The seconds abstraction is the provider, which manages and uses the sensor wrappers to collect data. The provider can use any of the available sensor wrappers to collect data through the common interface. This interface exposes the functionality to let a provider start and stop the data acquisition with a sensor wrapper, the functionality to discover the available sensor wrappers, and the possibility to send messages with the collected data from the sensor wrapper to the provider. When the provider application receives the collected data, it can process it as it desire.

Figure 4.4: Sensor wrapper used by one single application

This processing could in principle be anything, such as storing the data, analysing the data or forwarding the data. The data acquisition has been design to be as flexible as possible; any provider application using the common interface of sensor wrappers, can use them for collecting data. However, only one application can use each sensor wrapper at a time. As mentioned, the provider application can process the data in any way it desire. By creating a provider application that shares the data to other applications multiple applications can use the data from the same sensor wrapper simultaneously. The provider application could multiplex the received data to any application interested in the data, as illustrated in Figure 4.7.
Figure 4.5: Sharing the collected data between multiple applications

The data acquisition system is designed to be as flexible as possible. This is achieved by making as few assumptions as possible about the data sources and the applications that make use of the collected data. These applications that make use of the data, can either acquire it directly from the sensor wrapper application (as illustrated in Figure 4.6) or receive it from another provider application (as illustrated in Figure 4.7). In either case the sensor wrapper application collects and forwards the collected data. With the latter approach the provider application can forward the received data to multiple applications, which can use the data in any desired way. In the prototype developed as a part of this thesis, a provider application that stores the data to file or sends it to a server is created. An abstraction is added as in the multiplex example above, this abstraction stores the data to a file or forwards it to a server instead of forwarding it to other applications.
Chapter 5

Android specific design and implementation

In this section, the Android specific details of the design and implementation are described. This includes interprocess communication, how to find available sensor wrappers, starting and stopping data acquisition with these sensor wrappers, the data flow during data acquisition, and a description of the developed prototype applications. The code written as a part of this thesis is available at github: https://github.com/sveinpg/DMMS.

5.1 Interprocess communication

Inter-process communication (IPC) describes the mechanism of how separated components are communicating with each other. To increase security and privacy on modern smartphones, each application runs within its own process. Due to the separation of functionality into different applications interprocess communication is a crucial task in this project.

5.1.1 Interface between applications

From the high-level design we know that two types of messages are sent between the two application types. One of which are command messages sent between the two application types, and one message type for sending the collected data from the sensor wrappers to the provider. The command messages (hereafter referred to as commands) should be flexible and able to reach sensor wrappers unknown to the provider application to detect new sensor wrappers. However, the commands do not require as high performance as the message for sending the collected data. The commands sent between the two application types are:

- **HELLO**, this command is sent by the provider application to discover the available sensor wrapper applications on the mobile device.
• **REGISTER**, this command is sent as a response to the **HELLO** command by the available sensor wrappers.

• **START**, this command is sent to selected sensor wrappers (by the provider application) when the data acquisition should start. It contains information of where to send the collected data.

• **STOP**, this command is sent to the selected sensor wrappers (by the provider application) when the data acquisition should stop.

Unlike the command messages, the message for sending the collected data from the sensor wrapper to the provider application requires high performance. Otherwise, the this might become a bottleneck for the system. This message type is sent as a string (i.e. the JSON-object created by the sensor wrapper) from the sensor wrapper application to the provider application.

### 5.1.2 Comparing alternatives

In Android there are mainly three ways of doing interprocess communication:

• **Binder** [28], a remote-procedure-call (RPC) mechanism that enables a process to remotely invoke functions running on another process. Binder adopts a direct-message-copy scheme to transfer the IPC payload with only single data copy.

• **Intent** [31], a flexible message passing interface, implemented using two Binder IPC calls, allowing applications to send messages to each other.

• **ContentProvider** [42], a data storehouse mechanism that implements various SQL-like IPC interfaces, in which the query operation uniquely incorporates a shared memory region to facilitate the transmission of possibly large query results.

In [43] Hsieh et al. compare Binder, Intent and ContentProvider with respect to latency and resource usage. The experiments were conducted on a Samsung SGH-T959 Galaxy S running Android 2.2. Their experiments show that Binder performed better than Intent and ContentProvider with respect to latency, memory usage and CPU usage when the payload for the IPC was less than 4KB. While ContentProvider performed the best with payloads larger than 4KB. When the payload size exceeded 4KB the latency and CPU usage of the Binder increased significantly. This is because the serialized payload plus RPC headers is over the 4KB boundary of the initial kernel buffer which leads to repeated allocation and memory release of a temporary buffer. However, in newer mobile phones and versions of the Android operating system the size of kernel buffer has increased.
Considering the results of the discussed paper, and the fact that the payload between the applications typically is between 100 and 200 bytes, makes Binder the best suited alternative for sending the collected data to the provider application. By choosing to use Binder as the IPC mechanism, the best performance (and therefore minimal overhead) is ensured when sending such small strings between the applications. The Binder IPC can be implemented in two ways, either by using a Messenger or by implementing it using the Android Interface Definition Language (AIDL). A Messenger can receive one message synchronously at given time, while an AIDL implementation can handle multiple message simultaneously. Therefore, an AIDL implementation is chosen, to reduce the possibility of creating a bottleneck when sending the collected data to the provider application. For passing the command messages between the applications Intents are well fitted due to their flexibility; broadcasted Intents can be intercepted by everyone as long as they listen for it (by registering the correct intent-filter). The reduced performance of choosing Intents is negligible (since the command messages are small and infrequent) compared with the flexibility gained by being able to reach all applications on the mobile device.

5.1.3 Defining an AIDL Interface

To perform IPC using an AIDL [30] defined interface the interface needs to be defined in all applications involved in the IPC. The AIDL interface is defined in an .aidl file using Java syntax (see Listing 5.1), this file is placed in the src/ directory of both the application hosting the remote Service (the provider application) and any other application binding to the Service (the sensor wrappers). It is required that the .aidl file in all applications is identical (including the package-specification), otherwise the system will not recognize it as the same interface.

```java
package com.sensordroid;

interface IMainServiceConnection {
    /* Remote function */
    oneway void putJson(in String json);
}
```

Listing 5.1: AIDL interface between the two application types.

When building the application, the compiler generates code for the interface based on the interface defined in the .aidl file. The generated code compresses each method call, transfers it to the remote Service where it is decompressed and executed. However, the remote method needs to be implemented by the application developer of the provider application. This is done by creating a Service in the provider
application, by extending the interface (IMainServiceConnection) defined in the .aidl file. The interface contains an inner abstract class (Stub) that the provider application must extend and implement the functions defined by the AIDL interface (the putJson() method), as shown in Listing 5.2. Additionally, the onBind() method of the Service must be overridden to return the implemented Stub class. The implementation of the putJson() method is where the data collected by the sensor wrappers is received by the provider application, and the provider application can either process the received data in this method or pass it to another component for further computations.

```java
public class MainService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    private final IMainServiceConnection.Stub binder = new IMainServiceConnection.Stub{
        @Override
        public void putJson(String json) {
            /*
             * Receives string from a remote process and processes it
             */
        }
    }
}
```

Listing 5.2: Implementation of the AIDL interface

The client applications (the sensor wrappers) can then bind to this Service with a call to the bindService() method. The method specifies which Service to bind to, and when called the onBind() method of the specified Service is called. The onBind() method returns the implementation of the Stub class, which can be used to call the method defined by the AIDL interface. The callback from the the bindService() (the onServiceConnected() method) receives implemented Stub class as an argument (see Listing 5.3), which is then stored in a local variable named binder.

```java
IMainServiceConnection binder;
private ServiceConnection mConnection = new ServiceConnection()
```
// Called when the connection with the service is established
public void onServiceConnected(ComponentName className, IBinder iBinder) {
    // this gets an instance of the IRemoteInterface, which we can use to call on the service
    binder = IMainServiceConnection.Stub.asInterface(iBinder);
}

// Called when the connection with the service disconnects unexpectedly
public void onServiceDisconnected(ComponentName className) {
    Log.e(TAG, "Service has unexpectedly disconnected");
    binder = null;
}
};

Listing 5.3: Connection handling to the remote Service

The client application (the bound sensor wrapper) can now perform remote procedure calls by calling the remote methods with the IMainServiceConnection object (hereafter referred to as a proxy). The sensor wrapper applications can use this proxy to send the collected data as JSON-strings to the provider application, as shown in Listing 5.4 where a JSON string is sent to a provider application.

String message = new JSONObject().toString();
binder.putJson(message);

Listing 5.4: Sending a string to the remote Service

Calls from a remote process are dispatched from a thread pool the platform maintains on the provider application side. Multiple calls to the remote method from various bound clients (sensor wrappers) at the same time. The calls are submitted to the thread pool and might be executed simultaneously by different threads in the thread pool. Therefore, the implementation of the remote method (putJson()) is required to be thread-safe and resilient.

5.1.4 Sending and receiving broadcasts

To send command messages between two application types we have chosen to express the message as an Intent which is broadcasted. Broadcasted Intents are sent out to all BroadcastReceiver which subscribe to the specific Intent. The Intent defines an action for the Intent, which the BroadcastReceiver can use to subscribe
to the broadcast by defining an intent-filter (as shown in Listing 5.5).

```
<receiver
    android:name=".RespondReceiver"
    android:enabled="true"
    android:exported="true">
  <intent-filter>
    <action android:name="com.sensordroid.HELLO" />
  </intent-filter>
</receiver>
```

Listing 5.5: Manifest declaration of a BroadcastReceiver

The enabled option must be set to true for the receiver to be enabled and the exported option must be set to true to be reached by broadcasts outside the application, which is needed in our scenario with the separation of functionality into different applications.

Variables can be passed as a part of the Intent by appending them to it, with methods corresponding to the variable type. For example putStringExtra(), which is used to append a string to the Intent. The methods used for appending variables to the Intent takes two arguments, an identifier and a value. The identifier can be used to read the value from the Intent on the receiving side.

```
Intent helloIntent = new Intent();
helloIntent.setAction("com.sensordroid.HELLO");
helloIntent.putStringExtra("identifier", "string message");
sendBroadcast(helloIntent);
```

Listing 5.6: Sending an Intent

If the action of the broadcasted Intent match the action specified in the intent-filter of a BroadcastReceiver, the onReceive() method of the BroadcastReceiver is executed. This method performs the action of the Intent. For example, when receiving a START Intent the method starts the data acquisition with the sensor wrapper that received the Intent.

```
public class RespondReceiver extends BroadcastReceiver {
    private static final String TAG = "RespondReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "Got Intent");
    }
}
```
..Code to handle the intent.
*/
}
}

Listing 5.7: Receiving an Intent

The various command messages, and their corresponding action-string, is listed below.

HELLO - “com.sensordroid.HELLO”
Sent by the provider application to check for available sensor wrapper applications.

```
Intent helloIntent = new Intent("com.sensordroid.HELLO");
sendBroadcast(helloIntent);
```

Listing 5.8: Sending a HELLO-command as an Intent

REGISTER - “com.sensordroid.REGISTER”
Sent by the sensor wrapper applications as a response to the HELLO-command, to register as an available sensor wrapper. The name and ID of the responding sensor wrapper is appended to the Intent, as shown in Listing 5.9.

```
Intent registerIntent = new Intent("com.sensordroid.REGISTER");
registerIntent.putStringExtra("NAME", WrapperService.name);
registerIntent.putStringExtra("ID", context.getPackageName());
sendBroadcast(registerIntent);
```

Listing 5.9: Sending a REGISTER-command as an Intent

START - “com.sensordroid.START”
Sent by the provider application to start the selected sensor wrapper applications. A list containing the IDs of the selected sensor wrapper applications and information about which Service to bind to is appended to the message.
Intent start = new Intent("com.sensordroid.START");
start.putStringArrayListExtra("WRAPPERS", wrapper_list);
start.putExtra("SERVICE_ACTION",
    "com.sensordroid.service.START_SERVICE");
start.putExtra("SERVICE_PACKAGE", "com.sensordroid");
start.putExtra("SERVICE_NAME",
    "com.sensordroid.RemoteService");
sendBroadcast(start);

Listing 5.10: Sending a START-command as an Intent

STOP - “com.sensordroid.STOP”
Sent by the provider application to stop the selected sensor wrapper applications. A list of the selected sensor wrapper applications are appended, as for the START-command.

Intent stop = new Intent("com.sensordroid.STOP");
stop.putStringArrayListExtra("WRAPPERS", wrapper_list);
sendBroadcast(stop);

Listing 5.11: Sending a STOP-command as an Intent

5.1.5 Broadcast receivers
Hello receiver
A RespondReceiver must be implemented in all sensor wrapper applications. It subscribes to the HELLO-command, which is used to check which sensor wrapper applications are available. The sensor wrapper applications respond with a REGISTER-command containing their name.

public class RespondReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent respons = new Intent("com.sensordroid.REGISTER");
        respons.putExtra("NAME", WrapperService.name);
        respons.putExtra("ID", context.getPackageName());
        context.sendBroadcast(respons);
    }
}

Listing 5.12: Implementation of a RespondReceiver
Register receiver

A RegisterReceiver must be implemented in the provider application, it is subscribing to the REGISTER-command which is broadcasted by the sensor wrapper applications as a response to the HELLO-command. The name and the ID of the sensor wrapper application which broadcasted the Intent is appended to the it. Upon receiving the Intent the broadcast receiver passes name and ID of the sensor wrapper to the Activity responsible for listing the sensor wrapper applications using a broadcast, which is received by a local BroadcastReceiver in the Activity.

```java
public class RegisterReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Forwards the name to an Activity
        Intent i = new Intent("com.sensordroid.REGISTER");
        i.putExtra("ID", intent.getStringExtra("ID"));
        context.sendBroadcast(i);
    }
}
```

Listing 5.13: Implementation of a RegisterReceiver

Start receiver

A StartReceiver must be implemented in all of the sensor wrapper applications, it subscribes to the START-command which is used to start the data acquisition by the selected sensor wrappers. Before starting collecting data, the broadcast receiver should check if the sensor wrapper application is one of the chosen ones. To check this the list of IDs appended to the Intent is used, as shown in Listing 5.14.

```java
public class StartReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int wrapperId = -1;
        Bundle b = intent.getExtras();

        if(b!=null)
        {
            int counter = 0;
            // Check if the driver’s name is in the list.
            for (String elem : b.getStringArrayList("WRAPPERS")){
                if (elem.equals(context.getPackageName())){
                    Log.d(TAG, " id found");
                    wrapperId = counter;
                    break;
                }
            }
        }
    }
}
```

Listing 5.14: Implementation of a StartReceiver
Listing 5.14: Implementation of a StartReceiver

Stop receiver

A StopReceiver must be implemented in all of the sensor wrapper applications, it subscribes to the STOP-command which is used to stop the data acquisition by the selected sensor wrappers. Upon receiving the Intent the BroadcastReceiver checks if its ID is contained in the list appended to the Intent. If it is appended the BroadcastReceiver tells the WrapperService to stop acquisition.

```java
public class StopReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        /*
         * Check if the ID is contained in the appended list.
         */

        // Tells the WrapperService to stop the data acquisition
        Intent service = new Intent(context, WrapperService.class);
        service.putExtra("ACTION", "com.sensordroid.STOP");
        context.startService(service);
    }
}
```
5.2 Android components

The chosen high-level design and interprocess communication method places certain restrictions on which Android components the two application types needs. Firstly, each sensor wrapper is going to communicate with the data source in a separate thread in the background. In order to do this, each sensor wrapper should contain a Service (the WrapperService) to run in the background (i.e. not be dependant of a user interface). However, a Service does not run in a separate thread unless explicitly specified. Therefore, a designated thread is created within the Service to communicate with the data source. This Service binds to the remote Service (i.e. the Service of the provider application), which it sends the collected data to using the remote method implemented in the provider application.

Furthermore, each sensor wrapper contain three BroadcastReceivers to receive each of the three commands broadcasted by the provider application. The StartReceiver receives the START commands, the StopReceiver receives the STOP commands, and the RespondReceiver receives the HELLO commands. Each of these BroadcastReceivers uses an intent-filter to intercept the broadcasted Intents, and upon receiving an Intent the onReceive() method of the BroadcastReceiver is executed. The StartReceiver and StopReceiver tells the WrapperService to start or stop ac-
quiring data, while the RespondReceiver responds by sending a REGISTER command to the provider application. Each data source might have different configuration possibilities, such as setting the sampling frequency, or activating or deactivating certain data channels on the data source. Instead of exposing the configuration through the common interface (which could potentially limit the possible configuration possibilities), we chose to expose the configurations through a graphical user interface. Therefore, each sensor wrapper can optionally contain an Activity (SettingsActivity) for configuring the behaviour of the sensor wrapper.

The provider application also needs a BroadcastReceiver to receive the REGISTER command sent out by the RespondReceiver of the sensor wrapper. Furthermore by choosing to use Binder as the IPC mechanism, the provider application contains a Service (RemoteService) which all of the sensor wrappers can bind to. The RemoteService implements the remote method for receiving the collected data (as described in Section 5.2.3). Additionally, it would be natural (but not required) for the provider application to contain one or more Activities providing an user interface to the user. Typically, these Activities are the component that broadcasts the various commands to the sensor wrappers. One Activity could for instance present a list of the available sensor wrappers (by broadcasting a HELLO and list the sensor wrappers that respond), or expose buttons for starting and stopping the data acquisition of certain sensor wrappers.

5.3 Sensor wrapper discovery

As described in the previous chapter the provider application can check which sensor wrapper applications are available on the mobile device by doing a handshake with the sensor wrappers. To perform the handshake the commands defined in the previous section are used.

The handshake is initiated by a HELLO command sent out by the provider application. All of the sensor wrappers listens for the command by declaring a BroadcastReceiver (i.e. called RespondReceiver) with an intent-filter matching the action string of the command ("com.sensordroid.HELLO"). When the HELLO command is broadcasted the RespondReceiver's onReceive() method is called by the main thread of the sensor wrapper application. The method responds to the broadcast by broadcasting a REGISTER command (as shown in Listing 5.12). In the provided template project and the developed prototype, the name of the sensor wrapper is declared by the name variable in the WrapperService. The provider application listens for the broadcasted REGISTER command using a BroadcastReceiver with an intent-filter matching the action string of the REGISTER command (com.sensordroid.REGISTER). The name and ID of the
sensor wrapper that broadcasted the command is appended to the Intent, and can be fetched by calling `getStringExtra("NAME")` and `getStringExtra("ID")` on the Intent respectively. The provider application should now have received the names of all the available sensor wrappers, and is free to use them however it is suited. This could for instance be to list all of the available sensor wrapper applications, as done for the provider application developed for this thesis.

To ensure that each sensor wrapper application is identified by an unique ID we have chosen to make use of the package name [44] of each application. The package name of each application is declared in the manifest-file of the application and serves as an unique identifier for the application. Every application published to Android’s application market place has a unique package name. Therefore, we have chosen to use it as the ID for each sensor wrapper application. The package name of the BITalino sensor wrapper application developed in this thesis is "com.sensordroid.bitalino". The package name is structured as a reversed URL and may contain uppercase or lowercase letters ('A' through 'Z'), numbers, and underscores ('_').

### 5.4 Data acquisition

In this section the program flow during data acquisition is explained, from the moment the main application signals the data acquisition to start, during data acquisition, and until the moment the main application signals the data acquisition to stop.
5.4.1 Starting the data acquisition

The data acquisition is started by a START command from the main application. The ID of the selected sensor wrappers are appended to the Intent, as well as information about the Service the sensor wrapper should bind to. By sending the information about which Service to bind to the possibility of having multiple applications using the same sensor wrapper applications is made possible.

![Program flow when starting data acquisition](image)

Figure 5.3: Program flow when starting data acquisition

All of the sensor wrapper applications have a BroadcastReceiver (startReceiver) that listens for START commands by implementing an intent-filter as described in the interprocess communication section. When a START command is received the BroadcastReceiver checks the appended list for the ID of the sensor wrapper. If the ID is contained in the list, the sensor wrapper’s Service (WrapperService) is started with a start Intent, as shown in Listing 5.14. By the call to startService() the WrapperService is created and dispatched by the main thread of the sensor wrapper application. If the WrapperService is not yet created its onCreate() method is called. Regardless of whether or not the Service was already started, onStartCommand() is called next. Both START and STOP commands are passed to the WrapperService by calling startService(). The only difference between the two commands is the action, which is appended to the Intent used to start the Service. To determine if it is a start or stop command the type of the action is checked, in this case the type is a start action. Next, the Service checks if the data acquisition is already running by checking if the binder variable is null or not. If it is not null, the acquisition is running and nothing should be done. If the value is null the WrapperService starts the data acquisition by setting itself to the foreground and by binding to the remote Service specified in the Intent (i.e. stored in the service_action, service_package and service_name variables).
if(binder == null) {
    Intent service = new Intent(service_action);
    service.setComponent(new ComponentName(service_package,
        service_name));
    getApplicationContext().bindService(service,
        serviceConnection, Service.BIND_AUTO_CREATE);
}

Listing 5.16: Binding to the provider application

If the bind is successful the onServiceConnected() method in the serviceConnection is called which creates a new separate thread responsible for all communication with the data source. The communication thread connects to the data source, and collects data from it until it is told to stop by a call to its interrupt() method. Up until the onServiceConnected() call, all communication with the data source happens on the main thread. However, like mentioned earlier, it is undesirable to have long-running and or potentially blocking operations on the main thread (and the spawning of a new thread solves that problem). The main thread is responsible for creating and dispatching, hence no broadcasted command can be received if the main thread is blocked. Moreover, a STOP-command can not be received if the main thread is blocked.

class MainServiceConnection implements ServiceConnection {
    ...
    public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
        binder = IMainServiceConnection.Stub.asInterface(iBinder);
        connectionThread = new Thread(new ConnectionHandler(binder, name, driverId));
        connectionThread.start();
    }
}

Listing 5.17: Creating and starting the communication thread

When binding to the remote Service an IBinder is returned. This serves as a proxy and can be used to perform remote procedure calls, as discussed in Section 5.1.3.

5.4.2 Stopping the data acquisition

The data acquisition is stopped by a STOP command sent out by the provider application. The sensor wrapper selected to be stopped are
appended to this Intent, which allows the user to stop only the desired ones (as opposed to stopping all sensor wrapper applications).

![Figure 5.4: Program flow when stopping data acquisition](image)

The STOP command leads to the dispatching of the StopReceiver and its `onReceive()` method is called. Similar to the START command, the BroadcastReceiver checks if the sensor wrapper application is one of the sensor wrappers in the appended list. If the list contains the ID of the sensor wrapper application the Intent is passed to the WrapperService with a call to `startService()`.

```
1 Intent service = new Intent(context, WrapperService.class);
2 service.putExtra("ACTION", WrapperService.STOP_ACTION);
3 context.startService(service);
```

Listing 5.18: Telling the WrapperService to stop data acquisition

The WrapperService is started and the `onStartCommand()` is called. Since this method is called for both START and STOP commands, the action of the Intent. In this case it is a STOP command and if the WrapperService is collecting data from the data source it is stopped. To stop the data acquisition the communication thread is stopped by a call to its `interrupt()` method, telling it to disconnect from the data source. The communication thread frequently checks for interrupts in order to know when to stop collecting data.

```
1 public void stop() {
2     if(binder != null) {
3         try {
4             serviceConnection.interruptThread();
5             getApplicationContext().unbindService(serviceConnection);
6         }
7     }
8 }
```

86
Since the data acquisition is stopped, the sensor wrapper application no longer needs to send data to the provider application, so it unbinds from the RemoteService of the provider application. The binder variable is set to null to indicate that the data acquisition is stopped.

5.4.3 Data flow

In this section the data flow of the data acquisition is described, from the data source to the computations made with the collected data. As an example, this section discusses a provider application that forwards the collected data to a server using TCP. In Figure 5.5, the data flow from the data source to a server is illustrated. The data flow for other usages of the data is similar, differentiated by that collected data is used in another way instead of being sent to a server. As mentioned earlier, a sensor wrapper establishes a connection between the mobile device and the data source. The sensor wrapper application is bound to the RemoteService in the provider application, this Service implements the remote method (putJson(String json)) defined in the AIDL-interface, see Section 5.1.3. This remote method is made available to the sensor wrapper application through a proxy, which is stored in the binder variable. The remote method that is implemented in the prototype provider application in this thesis, either stores the String (i.e. the argument of the remote method) to a file or forwards it using TCP.
By sensing the analogous world the data source provides data, which is made available to the sensor wrapper application through the established connection. A separate thread (the communication thread) is running inside of the sensor wrapper application responsible for reading the values from the connected data source. When data is read from the data source, the communication thread submits a runnable DataHandler object (containing the collected data) to a thread pool for processing. To avoid data loss from buffer overflow on the data source the processing of the collected data is not done on the communication thread, see Section 4.6.3. When submitted to the thread pool the DataHandler object is placed in a queue, and is executed when one of the threads in the thread pool is free. The thread pool used in the implementation is an ThreadPoolExecutor [45] that uses a single thread operating off an unbound queue. The thread pool consists of one single thread to avoid packet reordering due to context switches. However, it is beneficial to use a thread pool instead of spawning a new thread, since the thread pool maintains a queue of the submitted tasks. When executed the run() method of the DataHandler object is invoked. This method processes the data (i.e. for values originating from the BITalino sensor board the processing is scaling the data with the transfer functions described in Section 2.2.5), before a JSON-object is created and sent to the provider application by a call to the remote method, see Listing 5.6. Since the Binder-interface only support marshalling of primitive types, the JSON-object is represented as a String when sent.
Listing 5.20: Creation and sending of JSON-object

When the remote method is called the method call (including its arguments) is compressed, copied to the provider application using a buffer, then the method call is reassembled and submitted to a thread pool in the provider application (maintained by the Binder-driver). When one of the threads in this thread pool is unoccupied the remote method call is executed. The implementation of the remote method might vary between each implementation of a provider application. In the provider application developed in this thesis the remote method creates a new runnable object of the received String and submits this to a thread pool maintained by the provider application.

Listing 5.21: Implementation of a runnable DispatchTCPHandler object

The type of this object depends on whether the collected data should
be sent to a server or stored to a file. In the case where the data is sent to a server a DispatchTCPHandler object is created, which writes the received String to a TCP-socket with a newline character appended (to separate the strings). In the case where the data is supposed to be stored to a file a DispatchFileHandler object, which writes the String (with a newline appended) to a file, is submitted to the thread pool. After execution of the DispatchTCPHandler or DispatchFileHandler the collected values have arrived at their destination, at the server or in the file. As mentioned earlier, the remote method needs to be thread-safe. Since the remote method only submits the received data to a thread pool in this implementation the thread-safe requirement is fulfilled.

```java
public class DispatchFileHandler implements Runnable {
    private final String frame; // String to send/
    private final FileWriter writer;

    public DispatchFileHandler(final String frame, FileWriter writer) {
        this.frame = frame;
        this.writer = writer;
    }

    /*
     * Sends the given string over the TCP-socket.
     */
    @Override
    public void run() {
        synchronized (writer) {
            try {
                writer.append(frame + "\n"); // Use newline to separate json-packets
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 5.22: Implementation of a runnable DispatchFileHandler object

The thread pool mentioned previously in this section is implemented as a FixedThreadPool, which is a fixed size pool of threads. The threads are reused to run the runnable class submitted to them. If all of the threads in the thread pool are currently occupied by a runnable object when a new object is submitted, the newly submitted runnable object is placed in a queue and executed as soon as one of the threads are free.
5.5 Provider application

The provider application developed in this thesis either saves the values collected by the sensor wrapper applications to a file, or sends them from the mobile device over the network using a TCP connection to an IP-address and port number specified by the user. The application presents the available sensor wrapper applications to the user, and provides the user the possibility to start and stop data collection from some or all of the sensor wrappers. The provider application writes the values received from the sensor wrappers to either a file or a socket of the users choice. To achieve this the provider application consists of three Activities, and hence three different user interfaces:

- **ListActivity.java**, presents a list of the available sensor wrappers to the user. It provides two buttons which each trigger a transition to one of the two other Activities.

- **SettingsActivity.java**, provides the user the possibility to configure if the collected data is written to file or to a socket. Additionally, it provides the possibility to specify the file name and IP port combination.

- **CollectActivity.java**, provides two buttons which allow the user to send `START` and `STOP` commands to sensor wrappers. Sending these commands will start and stop the data acquisition respectively.

![Figure 5.6: The Activities of the provider application](image)

In addition to these Activities the provider application consists of a RemoteService and a registerReceiver (described earlier in this chapter). In this section, each of the three Activities are explained in detail, starting with the **ListActivity** which is the Activity that starts when the application is launched.

5.5.1 ListActivity

Upon launch of the provider application the **ListActivity** is started. As for all Activities, it is started by a call to the **onCreate()** method
of the Activity. In the onCreate() method the user interface is created, by inflating the layout defined in activity_list.xml.

The layout contains a ListView to present the available sensor wrapper applications to the user, and two buttons to interact with the user. The available sensor wrapper applications are found as explained in Section 6.2, and are presented with a check box next to their name. When a REGISTER command is received by the registerReceiver, the name of the sensor wrapper application is passed to the ListActivity. The ListActivity then adds each name to an ArrayList, which contains the received names and is used to populate the ListView. Once the layout is inflated, onClickListeners are added to the buttons to detect if they are clicked. The SETTINGS and START COLLECTING buttons are used to change to the SettingsActivity and CollectActivity respectively. A click to the SETTINGS button will start the SettingsActivity.java, that presenting multiple options for customizing the behaviour of the application to the user. Such as choosing if the received data should be stored to a file or sent from the mobile device using a TCP connection.

```java
1 Intent configure = new Intent(ListActivity.this, SettingsActivity.class);
2 startActivity(configure);
```
Listing 5.23: Code to start the SettingsActivity

A click to the *START COLLECTING* button starts the **CollectActivity** with an Intent with the names of the selected sensor wrappers appended. The **CollectActivity** provides the user with the possibility to start and stop data acquisition from the sensor wrappers selected from the list presented in the **ListActivity** (as illustrated in Figure 6.6).

```java
1. Intent intent = new Intent(ListActivity.this, CollectActivity.class);
2. intent.putStringArrayListExtra("WRAPPERS", wrapperArrayList);
3. startActivity(intent);
```

Listing 5.24: Code to start the CollectActivity with the selected sensor wrappers

The list of selected sensor wrappers (`wrapperArrayList`) is generated when the *START COLLECTING* button is pressed. The list is generated by appending the *ID* of all sensor wrappers that have a checked check box next to it in the **ListView**.

### 5.5.2 SettingsActivity

The **SettingsActivity.java** is launched when the user clicks on the **SETTINGS** button in the **ListActivity.java**. This Activity is responsible for presenting the settings to the user and saving them to shared preferences to preserve them between launches. Shared preferences are used to store key-value pairs, which can be accessed from anywhere in the application. As for the **ListActivity**, the layout corresponding to the **SettingsActivity** (defined in `settings_layout.xml`) is inflated when the Activity is started.

The layout includes two **ToggleButton**s, one to choose whether to write to file or send the data packets over the network, and one to choose between TCP or UDP when sending the data. It also has editable **text fields** (**EditText**), where the user can specify the filename, IP-address and port number to use and a **CheckBox** to choose whether to update the package count **text field** in the **CollectActivity** (see Section 5.5.3). Updating the packet count involves sending an **Intent**, which uses the same buffer as the remote method call to perform the broadcast. Therefore, the user is given the choice to retain from updating the package count, in order to reduce the usage of the buffer. A button to save the settings entered by the user is located at the bottom of the layout. When the layout is inflated, the previously saved values are
read from the shared preferences and inserted into the corresponding layout objects, as shown in Figure 5.8.

### Saving settings

When the button for saving the current values is clicked, the values entered by the user are collected from the layout objects and saved to the shared preferences. Each value in the shared preferences is identified by a *key*, which can be used to overwrite and read the already saved value. When saving the values to the shared preferences the value can be written by calling `apply()` or `commit()` on the `SharedPreferences` object. The `apply()` method will write to disk asynchronously, while `commit()` does it synchronously [46]. Since the main thread should not be blocked, and the execution of the Activity is running on the main thread, the `apply()` method is used to avoid blocking the main thread.

```java
SharedPreferences sharedPref = 
    getActivity().getSharedPreferences(
        sharedKey, Context.MODE_PRIVATE);
String value = mEditFileName.getText().toString()
sharedPreferences.edit().putString(fileNameKey, 
    value).apply();
```
Listing 5.25: Code for saving values to SharedPreferences

When the values are saved to the SharedPreferences, the configuration is considered finished and the ListActivity is started again with an Intent.

5.5.3 CollectActivity

When the user clicks the START COLLECTING button in the ListActivity the CollectActivity is started. The Activity is dispatched by the main thread of the provider application by a call to the onCreate() method of the Activity, which inflates the layout.

Figure 5.9: Screenshot of the CollectActivity

The layout consists of two buttons for starting and stopping the data acquisition, as well as a text field displaying the number of sent packets (if the appropriate settings is selected). The text field is updated by broadcasts from the RemoteService. This is done because the UI toolkit is not thread safe and should only be used by the main thread.

When the layout is inflated the provider application prepares to receive data from the sensor wrapper applications. As a part of this preparation the CollectActivity starts the RemoteService by binding to it.

RemoteService

As for Activities the onCreate() method is called the first time a Service is started. When the RemoteService is created, the
settings saved to the shared preferences are read. Depending on the selected settings either a file with the selected name is opened, or a socket is created to the selected IP-address and port combination. Creating a socket involves blocking calls and should not be executed on the main thread. To overcome this the connection is established using an ASyncTask (i.e. executed on a separate thread). In the case where it is selected to store the data to a file, a file with the selected filename is created in the Download folder in the external storage. By saving the file to the external storage it is made available to other applications, as described in Section 3.5.2. After opening the file, or socket, the RemoteService is set to the foreground to increase the rank of the process. Details about this can be found in Section 5.7.3. Once the Service is started in the foreground, the Service is ready to perform the remote method calls from the sensor wrapper applications. For this implementation of the provider application the putJSON() method submits a Runnable object, containing the String received as the argument, to a thread pool. The type of the runnable object depends on the choice of whether to use file or network. In the case where file is selected a DispatchFileHandler is submitted, and a in the case where network is selected a DistpatchTCPHandler is submitted. The thread pool queues the submitted objects, and the first object in the queue is executed as soon as a thread is free. The DispatchFileHandler and DispatchTCPHandler writes to the opened file or to the created socket respectively.

Buttons

An onClickListener is created for each of the buttons in the layout.

- When the start-button is clicked the CollectActivity sends out a START command to all the selected sensor wrappers, as described in Section 5.1.4. Additionally, the background color of the layout is changed to green to indicate that the data acquisition is started.

- When the stop-button is clicked the CollectActivity sends out a STOP command to all selected sensor wrappers, as described in Section 5.1.4. The background color of the layout is changed to red to indicate that the data acquisition is stopped.

It is possible to change from the CollectActivity back to the ListActivity by pressing the back-button of the mobile device. To customize the behaviour of the application when the back button is pressed the onBackButtonPressed() method of the Activity is modified. When the user presses the back button a STOP command is sent to the sensor wrapper applications and the ListActivity is started with an Intent.
5.6 BITalino sensor wrapper application

In this section, a sensor wrapper application developed for collecting data from a BITalino sensor board is described. The sensor wrapper application is tailored for performing sleep monitoring. The user interface of the sensor wrapper is described, before the adoption to Link Layer technology and communication protocol used by the BITalino is described in detail. The BITalino board consists of six analog input channels to collect data at one of four different sampling frequencies (1, 10, 100, or 1000 Hz). A range of different sensors can be plugged in to the analog inputs. Depending on the type of sensor connected to an analog channel, different transfer functions are used to scale the collected values from the channel, as described in Section 2.2.5. An user interface is presented to the user by the BitalinoActivity, providing the user the possibility to customize which analog input channels to use, which transform function to use for each channel, and to specify the sampling frequency.

5.6.1 Development with BITalino

The BITalino team offers multiple software development kits (SDK) to communicate with the BITalino sensor board [4] for various programming languages. The application described in this section was developed using the Java SDK provided by the BITalino team. During the development of the application some bugs and shortcomings in the Java SDK were discovered. The solution to these issues were merged with the provided SDK and the contributions to the SDK from this thesis are covered in Appendix B.

5.6.2 User interface

The user interface of a sensor wrapper is used for configuring the data acquisition from the specific data source. For the BITalino sensor wrapper the user interaction is defined in the BITalinoActivity, which inflates the layout defined in wrapper_layout.xml. The user interface consists of a drop-down menu for each of the analog input channels, and editable text fields for giving a description for each analog input channel. It also includes the option to specify the sampling frequency and the MAC-address of the BITalino device to use.

From the drop-down menus the user can turn on or off data collection for each of the analog input channels, and specify which transfer function to use for scaling the collected values for each analog channel. The first drop-down menu is associated with analog channel A1 on the BITalino sensor board, the second to A2, etc., all the way up to A6. Underneath each of the drop-down menus there is an editable text field to add a description to the analog input channel. Underneath the drop-down menus there is a SeekBar to set the sampling frequency, an
editable text field to specify the MAC-address of the BITalino device, and a button which saves the entered values.

When the BitalinoActivity is launched (and the layout is inflated) the previously saved settings are read from shared preferences and put into the corresponding layout object as described above. When the button on the bottom of the layout is pressed, the values entered in the layout objects are written to shared preferences, which makes them accessible from anywhere in the application.

5.6.3 Collecting data

In addition to the BitalinoActivity the sensor wrapper application has a service (WrapperService) and three broadcast receivers (RespondReceiver, StartReceiver and StopReceiver) as described previously in this chapter. When a START command is received (by the StartReceiver), the WrapperService is started and binds to the remote service specified in the START command (as described in Section 5.4). When the WrapperService has bound to the remote service, the onServiceConnected() method is called a thread (CommunicationHandler) to communicate with the data source is started. This thread collects data from the data sources and sends the collected data to the bound provider application. In order to avoid buffer overflow (as described in Section 4.6.3) the communication thread contains a thread pool, which it passes the computational work to. This includes extracting and sending the data to the provider application.

```
public void onServiceConnected(ComponentName componentName,
   IBinder iBinder) {
```
When the services are bound successfully an IBinder object is returned, hereafter refereed to as a proxy. This object exposes the interface to the remote service, which includes the method (putJson()) for sending a string from the sensor wrapper to the bound provider application. The proxy to the remote service is stored in the binder variable. The proxy is then passed to the communication thread enabling it to send strings to the provider application. When the communication thread is started the run() method of the CommunicationHandler is called (see Listing 5.27), which first sends the metadata associated with the data source (i.e. executing the sendMetadata() method), then establishes a Bluetooth connection to the BITalino board (i.e. executing the connect() method), and then starts collecting data (i.e. executing the collectData() method). It continues collecting data until it receives a STOP command from the provider application, which prompts it to stop the data acquisition and close the established Bluetooth connection (by executing the resetConnection() method).

Listing 5.27: Starting the thread for communicating with the data source

```
public void run() {
    ...
    sendMetadata();
    if (connect()) {
        collectData();
        resetConnection();
    }
    ...
}
```

Listing 5.27: Main actions of the communication thread

Listing 5.27 shows the main actions of the communication thread. However, the code snippet is placed inside a while-loop to provide a Bluetooth reconnection mechanism (as described in Section 5.6.4).
Sending metadata

When the communication thread is started, the `sendMetadata()` method is called. This method creates (and submits it to the thread pool) a runnable `MetadataHandler` object containing the metadata associated with the specific data source. When run, the `MetadataHandler` creates a `JSONObject` and sends it to the provider application using the proxy. The metadata consists of the name and ID of the sensor wrapper, as well as an array containing the data type, metric and the description of each data channel (as described in Section 5.28). The name is specified by the developer of the sensor wrapper application, the ID is obtained by the position in the sensor wrapper list received by the `StartReceiver`, and the information about each data channel is obtained based on the configuration selected by the user in the `BITalinoActivity`. Moreover, the information about each data channel is read from the `SharedPreferences`.

```
JSONObject res = new JSONObject();
try {
    res.put("type", "meta");
    res.put("name", name);
    res.put("ID", ID);

    JSONArray channels = new JSONArray();
    for (int i = 0; i < ids.length; i++) {
        JSONObject element = new JSONObject();
        element.put("id", ids[i]);
        element.put("type", dataTypes[i]);
        element.put("metric", metrics[i]);
        element.put("description", descriptions[i]);
        channels.put(element);
    }
    res.put("channels", channels);
} catch (JSONException e) {
    e.printStackTrace();
}
...
 binder.putJson(res.toString());
```

Listing 5.28: Method for creating and sending a metadata packet

To represent the metadata a `JSONObject` is created, consisting of key-value pairs. The keys are used to set and retrieve the values associated with the key. The "type"-key is given the value "meta" to indicate that the current `JSONObject` is a meta data packet. Then, the name, ID, and data channel information array is appended to the object with "name", "id" and "channels" respectively as their keys. This results in a `JSONObject` (res) as the one shown in Figure 5.3. The `JSONObject` is then converted to a string and sent to the
provider application by using the proxy to the remote method, by calling
binder.putJson(res.toString()).

Connecting to the data source
The connection to the data source is implemented in the connect() method located in CommunicationHandler. The connect() method establishes a Bluetooth connection to the BITalino device. Afterwards a BITalinoDevice object is created with the sampling frequency and analog channel combination selected by the user in the BITalinoActivity.

```java
/* Creates a bluetooth connection */
public void connect() throws IOException, BITalinoException {
    final BluetoothAdapter blueAdapt = BluetoothAdapter.getDefaultAdapter();
    final BluetoothDevice dev = blueAdapt.getRemoteDevice(remoteDevice);
    final List<UUID> uuidList = new ArrayList<>();
    ParcelUuid[] uuidParcel = dev.getUuids();
    boolean connected = false;
    for (ParcelUuid uuid : uuidParcel) {
        BluetoothSocket tmp;
        try {
            tmp = dev.createInsecureRfcommSocketToServiceRecord(uuid.getUuid());
        } catch (IOException ioe){
            ioe.printStackTrace();
            continue;
        }
        mSocket = tmp;
    blueAdapt.cancelDiscovery();
    try {
        mSocket.connect();
        connected = true;
        break;
    } catch (IOException ioe){
        ioe.printStackTrace();
    }
}
if(!connected){
    throw new IOException("Could not connect to bluetooth device");
}
bitalino = new BITalinoDevice(SAMPLING_FREQ, ids);
bitalino.open(mSocket.getInputStream(), mSocket.getOutputStream());
```
Listing 5.29: Method for connecting to the BITalino device

The connect() method collects the UUIDs of the paired Bluetooth devices by calling getUuids() on the BluetoothDevice object. The UUIDs are then used one by one to try and connect to the Bluetooth device (with the MAC-address fetched from the shared preferences) until a connection is established. If the connection is successful a BITalinoDevice is created and opened to specify the InputStream and OutputStream used for communicating with the BITalino board. When creating a BITalinoDevice the analog channels and sampling frequency to use are specified to be the values fetched from SharedPreferences. At this point, the mobile device is connected to the BITalino board which is in the Idle state waiting for further instructions from the connected mobile device.

Collecting data from the data source

When the Bluetooth connection is established the collectData() method is called. The method reads data packets from the BITalino until a STOP command is received from the provider application. When this method is entered the BITalino board is in the Idle state, waiting for further instructions from the connected computing device. The mobile device sends a command to the BITalino board, by calling the start() method of the BITalinoDevice, telling it to change from the Idle state to the Live state. In the Live state the BITalino board collects data values from the selected analogs input at the sampling frequency specified by the user. To read the values from the BITalino board the bitalino.read(int frames) method is called, which collects a data packet from the InputStream and creates a BITalinoFrame (frame) object of it. A BITalinoFrame is an object that holds the information contained in the data packet received from the BITalino board. A runnable FrameHandler object is created for each of the BITalinoFrames, this object is submitted to the thread pool to always keep the communication thread free to avoid losing data due to buffer overflow. The FrameHandler object extracts and creates a JSONObject of the data values in the BITalinoFrame, which it then sends to the bound provider application.

```java
private void collectData()
{
    try {
        bitalino.start();

        while (!interrupted) {
            if (Thread.currentThread().isInterrupted()) {
                interrupted = true;
                return;
            }
```

102
Listing 5.30: Method for collecting data values from the BITalino board

**Extracting data**

The next step is to rewrite the data passed to the FrameHandler before sending it to the provider application. In this context rewriting the data means extracting the sampled values from the data-object and creating a JSONObject of them. As stated above, the data object used to pass the data collected from the BITalino is a BITalinoFrame. Extracting values from the BITalinoFrame can be done by calling the getAnalog(int pos) method on the BITalinoFrame object. Different scaling functions (to convert the data values to the proper unit) are applied to the data from each channel depending on the data type chosen in the drop-down menus. This is done with a switch statement which switches on the chosen position in the drop-down menu for each of the data channels. Information about the selected position is available in the int array typeList, which is read from the SharedPreferences.

```java
Double[] data = new Double[channelList.length];
int index = 0;
for (int type : typeList){
    if (index >= channelList.length)
        break;
    switch (type) {
        case BitalinoTransfer.TYPE_RAW:
            data[index] = Double.valueOf(frame.getAnalog(channelList[index++]));
            break;
        case BitalinoTransfer.TYPE_LUX:
            data[index] = SensorDataConverter.scaleLuminosity(channelList[index], frame.getAnalog(channelList[index++]));
            break;
    }
    index ++;
}
```
Listing 5.31: Method for extracting and sending the collected data

When a data value is scaled using the appropriate transfer function, the value is appended to an array containing the data values. When all data values are scaled, a JSONObject is created as described in Section 4.3.3. The JSONObject is then converted to a string and sent to the provider application, using the proxy to the remote service.

Cleaning up the connections

When the data acquisition is stopped by a STOP command from the provider application, the data acquisition stops and the collectData() method returns. Then the resetConnection() method is called to clean up the connections opened in the connect() method. This includes stopping data acquisition from the BITalino device by calling stop() on it, and closing the Bluetooth socket used for communication.

Listing 5.32: Method for cleaning up the connections to the data source
5.6.4 Reconnection

One of the intended usages for the whole system and especially the BITalino sensor wrapper is to monitor certain bodily functions of humans in their own home while sleeping, with the BITalino sensor board attached to the person being monitored. To do this the mobile phone and BITalino need to be within bluetooth range of each other.

Problem

One potential problem would be if the person gets up from bed and moves to another location in the home, such as visiting the toilet. This could lead to disconnection between the mobile phone and the BITalino device because the bluetooth devices are not within the bluetooth range of each other, as shown below. This makes it impossible to collect acquire data while the devices are disconnected.

To cope with such situations we want the connection to be reestablished “as soon as possible”. In other words the connection should be reestablished when the BITalino and mobile phone are again within their bluetooth range.

Solution

Ideally, the reconnection should happen at exactly the moment when the devices are just within range of each other again. For this to happen, we need to know when they are within bluetooth range. Unfortunately the Application framework [47] does not provide a way to determine if the devices are within range of each other. The only way to know if a paired bluetooth device is within range is by attempting to establish a connection with it. If the attempted connection was unsuccessful we can conclude that the devices are most likely outside each other range. On the other hand if the attempt was successful, the devices are connected.

To reconnect to the BITalino device, a mechanism which periodically attempts to reconnect to the device is implemented. Between each attempt the thread responsible for the connection sleeps for a given period of time. If the connection was successful the data acquisition starts. If the connection attempt was unsuccessful the thread sleeps for a period before a new attempt is made. This procedure continues until the sensor wrapper application has been stopped by STOP signal from the provider application.

Power consumption

A way to reduce the power consumption associated with the reconnection mechanism is to wait a period of time between each reconnection attempt. This would be a trade off between power consumption and reconnection time. Longer periods between each reconnection attempt could
lead to longer reconnection time, for instance if the BITalino device re-enters bluetooth range in the start of a sleep period. Hence a reconnection delay as large as the sleep period could occur.

**Implementation**

The implementation below attempts to connect to the BITalino device as long as the thread does not get interrupted. The thread gets interrupted when the user press the STOP button in the provider application. The sleep period between each reconnection attempt is initially one second. Each time a connection attempt is unsuccessful the sleep period is multiplied by two as long as the sleep period is under 30 seconds.

```java
int sleepTime = 1000;
while (!interrupted) {
    if (Thread.currentThread().isInterrupted()) {
        interrupted = true;
        return;
    }
    try {
        if (connect()) {
            sleepTime = 1000;
            collectData();
        }
    } catch (InterruptedException ie) {
        ie.printStackTrace();
        return;
    }
    try {
        Thread.sleep(sleepTime);
        if (sleepTime < 30000) {
            sleepTime = sleepTime*2;
        } else if (sleepTime > 30000) {
            sleepTime = 30000;
        }
    } catch (InterruptedException ie) {
        ie.printStackTrace();
        return;
    }
    resetConnection();
    try {
        Thread.sleep(sleepTime);
        if (sleepTime < 30000) {
            sleepTime = sleepTime*2;
        } else if (sleepTime > 30000) {
            sleepTime = 30000;
        }
    } catch (InterruptedException ie) {
        ie.printStackTrace();
        return;
    }
}
```

Listing 5.33: Reconnection mechanism for the BITalino sensor wrapper

### 5.7 Collecting data over a long period

Collecting data over a long period of time can introduce problems. Connections might be lost, applications might get killed. In this section the potential problems and the implementation of the corresponding workarounds are discussed.
There are mainly two reasons why an application gets killed. The operating system enters sleep mode to save power or the applications priority gets too low and the application is killed to reclaim resources.

### 5.7.1 Sleep

To prolong battery time Android devices first dim, then turn off the screen, before turning down the frequency of the CPU. WakeLocks can be used to keep the CPU running, to prevent the screen to dim and turn off. Any application using a WakeLock must request the WakeLock permission in the application’s manifest.

```
<uses-permission
    android:name="android.permission.WAKE_LOCK" />
```

Listing 5.34: Acquiring permission to use WakeLocks

For the applications developed for this thesis, it is sufficient to keep the CPU running. The user will probably not be interested in watching the packet count for a long period. Since it is desired to keep the CPU running even when the screen is off a partial WakeLock is used. Each application could possibly acquire a WakeLock to keep the CPU running. However, one WakeLock is enough to keep all of the application running; only the provider application needs to acquire a WakeLock. The WakeLock is required when the system is acquiring data, namely between onCreate() and onDestroy() of the provider applications RemoteService.

```
PowerManager pm = (PowerManager)
    getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl;

onCreate(){
    wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
        "RemoteService");
    wl.acquire();
}

onDestroy(){
    wl.release();
}
```

Listing 5.35: Acquiring a WakeLock

For Android versions prior to Android 6.0 the partial WakeLock is sufficient to keep the CPU running. However, in Android 6.0 Doze was introduced, which overrides WakeLocks.
5.7.2 Doze

In Android 6.0, also known as marshmallow, Doze was introduced. To save power Doze suspends applications when the device is not in use, as described in Section 3.6. Three requirements need to be fulfilled to enter Doze mode.

• The device needs to be unplugged.

• The device needs to be stationary.

• The screen needs to be turned off.

Preventing doze mode

Breaking one of the three requirements is enough to not enter doze state. Hence plugging in the power, keeping the phone in motion, or keeping the screen on will prevent the device from entering the doze mode. Keeping the phone in motion is not always possible, and would be a poor solution to this problem. While both keeping the screen on and the power plugged in are better alternatives.

• The screen could be kept on by using a WakeLock. There are four different WakeLocks, while only three of them would break one of the requirements. Holding a PARTIAL_WAKE_LOCK is not sufficient to prevent doze mode. This is because it does not keep the screen on, which the three other WakeLocks would do. If the user uses the power button it will override the WakeLock and the screen would be turned off, leaving the device in doze risk again.

• Keeping the power cable connected also prevents the device from entering the doze mode. This would only prevent the device from entering the doze mode and a PARTIAL_WAKE_LOCK should still be used to keep the CPU running if the screen turns off.

5.7.3 Priority

Android might decide to shut down a process at some point in time, e.a. when memory is low and required by other processes that got a higher rank. Application components running in the process that is killed are consequently destroyed. As an effort to prevent this the process can be set to the foreground decreasing the possibility for it to be killed, when the applications are collecting data.

Moreover data is acquired between onCreate() and onDestroy() of the RemoteService for the provider application, and as long as the sensor wrappers applications are bound to the main application. The code in Listing is executed in the onServiceConnected() of the provider applications RemoteService, and in the onServiceConnected() method in the sensor wrappers. The onServiceConnected() method is called when the sensor wrapper is successfully bound to the provider application. The user is notified with a notification when a process is given
foreground priority, this can be handy to show the user which driver applications that are running as well.

```java
// Create notification
final NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
builder.setSmallIcon(R.drawable.stat_notify_chat);
builder.setContentTitle("BITalino");
builder.setTicker("Starting");
builder.setContentText("Collecting data");

Intent i = new Intent(this, WrapperService.class);
i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
final Notification note = builder.build();

// Give the Service foreground priority.
startForeground(android.os.Process.myPid(), note);
```

Listing 5.36: Setting an Android component to the foreground

To give the service default priority again `stopForeground(boolean)` is called. This is done by the provider application when the `RemoteService`'s `onDestroy()` method is called, and by the sensor wrapper application when it unbinds from the provider application.

```java
stopForeground(true);
```

Listing 5.37: Setting the rank of an Android component back to default
Part III

Evaluation
Chapter 6

Experiments

In the requirement analysis in Section 4.1 certain requirements are specified. Some of these requirements, like the extendibility for new sensors and that data is represented in an understandable way, are satisfied through argumentation in the presented design, see Chapter 4 and 5. When it comes to the requirements not covered by argumentation in Chapter 4 and 5, experiments are conducted to convince the reader that they are also satisfied. These requirements are the following: the functionality to collect data, that the system is resilient, with respect to data loss and robustness, and lastly that the resource usage of the system is reasonable. Three experiments were conducted to show that the mentioned requirements were satisfied. To show that the system is resilient and able to collect data an overnight experiment was conducted, in the experiment the system collected data during a whole night. By being able to collect data without data loss during a whole night we consider the system to be resilient. To further evaluate the robustness of the system, some simple tests were performed to show that the bluetooth reconnection mechanism of the BITalino sensor wrapper work as intended. Lastly, a set of experiments measuring the CPU usage of the system while collecting data from a BITalino sensor board were performed to evaluate the resource usage of the data acquisition system.

In this chapter, we present how the testing and evaluation of the applications were conducted, as well as the results of the experiments. The goal of the experiments is to measure performance of the applications developed as a part of this thesis, and to show that the functional requirements presented in the design and implementation chapter are satisfied. Since the BITalino sensor wrapper was developed as a prototype for sleep monitoring, the experiment scenarios resemble actual sleep monitoring at home. All the experiments were performed using an Asus Nexus 7 tablet and a BITalino plugged kit. The specifications of both the devices are shown in Table 6.1.
### Table 6.1: Device specifications and settings

To get as accurate and consistent results as possible the mobile device was restarted, put in to airplane mode with bluetooth turned on, and fully charged before each run of all the three experiments. Moreover, this was done to create equivalent experiment environments for all runs of each experiment.

### 6.1 Measuring the CPU-usage

The experiment was conducted by collecting data with a BITalino sensor board using different sampling frequencies. Most modern mobile devices use a CPU with multiple cores, where the measured CPU-usage is given as a combination of the process time spent on all of the cores combined. The CPU-usage is measured as a percentage, and is calculated by summing the number of clock ticks used by each application on each of the cores since the last measurement and dividing it by the total number of clock ticks on all cores since last measurement. For instance, if the last measurement was 1000 clock ticks ago on a CPU with two cores, and during that time process 1 used 500 clock ticks on one of the cores, then process 1 used 25% of the CPU. To measure the CPU-usage of an application the Android Debug Bridge (ADB)[48] was used, which calculates the CPU-usage as explained above. In order to use ADB to measure the CPU-usage the mobile device was connected to the computer using a USB cable.

For this experiment the average of the measured CPU-usages of each run are used to evaluate the results.

#### 6.1.1 Android Debug Bridge

ADB provides the utility to dump the CPU-usage of the whole system and each of the processes to a terminal. This is done by using the top
command (adb shell top) which periodically outputs the CPU-usage of the whole system and the top processes. The ADB top command rounds the total and per process CPU-usage to the closest integer; therefore, to increase the precision of the measurements, BusyBox [49] is used instead. BusyBox calculates the CPU-usage in the same way as the adb top command, with the difference that the CPU-usage is given as a decimal number instead of an integer resulting in a higher precision. BusyBox periodically prints an output, as the one in Listing 6.1, containing information about the CPU-usage of each process and all of the processes combined. In this experiment such an output is produced every fifth second, which makes the measurements an average over the last five seconds.

Listing 6.1: Sample output of CPU measurements using Busybox

At line 4 in Listing 6.1 we can see that the user applications use 37.1% of the CPU and that the system uses 9.9%. In the subsequent lines the CPU-usage per process is listed. Moreover com.sensordroid.bitalino, which is the BITalino sensor wrapper, uses 30.6% of the CPU, and com.sensordroid, the provider application, uses 12.9%. When measuring the CPU-usage on a mobile device some of the CPU-usage resides from the measurement itself, such as top and busybox in Listing 6.1. To get a more accurate result, we use a python script to automatically extract the relevant information from the outputted statistics.

6.1.2 Parsing the output file

By extracting and adding the percentages on the second line the total CPU-usage of the system is obtained. In the Python code in Listing
6.2, the second line (i.e. the fourth line in Listing 6.1) of the BusyBox output is located and summed up. The subsequent lines of the output are then parsed, and CPU-usage imposed by the measurements is subtracted from the total CPU-usage. After subtracting the measurement-imposed CPU-usage the result is appended to a list of the measurements of CPU-usage. Once all measurements are performed and all data is stored in the list the following statistics are calculated: maximum, minimum, average and standard deviation.

```python
cpu_usage = []
sum_usg = 0
first = True

with open(sys.argv[i+2]) as f:
    for line in f:
        splitted_line = line.split()

        if splitted_line[0] == "Mem:":  # New batch
            if first == False:
                cpu_usage[i].append(sum_usg)
                sum_usg = 0
            first = False

        if splitted_line[0] == "CPU:":  # Total CPU-usage
            usr_usg = float(splitted_line[1].split('%')[0])
            sys_usg = float(splitted_line[3].split('%')[0])
            sum_usg += (usr_usg + sys_usg)

        if len(splitted_line) < 9:
            continue
            if splitted_line[8] == "busybox" or splitted_line[8] == "top" or \
               splitted_line[8] == "logcat" or splitted_line[8] == "/sbin/adbd":  # CPU-usage due to monitoring
                sum_usg -= float(splitted_line[7])

minimum = min(cpu_usage)
maximum = max(cpu_usage)
average = sum(cpu_usage)/len(cpu_usage)
tmp = 0
for x in cpu_usage:
    tmp += (x−averge)^2
standard_dev = tmp/(len(cpu_usage)−1)
```

Listing 6.2: Python script parsing the BusyBox output
6.1.3 Scenarios

As explained in the design and implementation, the data acquisition system consists of a provider application and possibly multiple sensor wrapper applications. The software separation into different applications introduces some overhead from interprocess communication. In this experiment, two scenarios are used to measure the CPU-usage of the system. One of the scenarios is used to measure the total CPU-usage when collecting and storing data, and the other scenario is used to measure how much of the CPU-usage in the first scenario that was imposed by collecting the data from the BITalino sensor board. The first scenario measures the CPU-usage when collecting data with the BITalino sensor wrapper, sending the data to the provider application which then writes the data to file. The second scenario also collects data with the BITalino sensor wrapper, but does not send it to the provider application. The difference between these two scenarios should be the CPU-usage of sending the data over IPC and writing it to file. The first scenario will hereafter be referred to as the file scenario, and the second one as the collect scenario. When measuring the CPU-usage of file scenario, a part of the CPU-usage stems from the BITalino sensor wrapper, bluetooth drivers and various background processes in the operating system. This part is measured in the collect scenario, and compared to the measured CPU-usage of the file scenario in the results.

Configurations

Both the scenarios uses the BITalino sensor wrapper and provider application as described in the design and implementation, and computing devices with the setup described in Table 6.1. Test runs of the showed very little variation in the CPU-usage over time and between runs, therefore 3 runs of 5 minutes for each of the four sampling frequencies (1, 10, 100 and 1000 Hz) were for performed for each scenario.

6.1.4 Results

In this section, the results of the file scenario and collect scenario are presented, analysed and discussed. The average of the 3 runs for each sampling frequency is presented in a table, while the raw data for each run is presented in a graph. The results are presented for each of the sampling frequencies, starting with the results for 1 Hz.

1 Hz

In both the file scenario and the collect scenario the average CPU-usage is below 2 percentage. The average CPU-usage for the collect scenario is actually marginally higher than for the file scenario with 1.292016 % versus 1.1298636% for the collect scenario. However, this is most likely
due higher CPU-usage by the background processes in the operating system.

<table>
<thead>
<tr>
<th>Metric</th>
<th>File scenario</th>
<th>Collect scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0.166667</td>
<td>0.700000</td>
</tr>
<tr>
<td>Max</td>
<td>4.966667</td>
<td>3.666667</td>
</tr>
<tr>
<td>Average</td>
<td>1.292016</td>
<td>1.1298636</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.540636</td>
<td>0.292235</td>
</tr>
</tbody>
</table>

Table 6.2: Average statistics of the measured CPU-usage for 1 Hz

From the plots of the raw data in Figure 6.1 it is clear that the CPU-usage is both low and stable for both scenarios, with oscillations between 0.166667% and 4.966667%.

![CPU-usage plots](image)

(a) CPU-usage of file scenario  (b) CPU-usage of collect scenario

Figure 6.1: CPU-usage for all runs with a sampling frequency of 1 Hz

10 Hz

When collecting data at 10 Hz the CPU-usage has increased compared with the results for collection at 1 Hz. The average CPU-usage for the file scenario is now just below 5%, while the collect scenario uses 3.5% of the CPU in average. At this sampling rate the data collecting has a bigger impact on the total CPU-usage of the mobile device, but is still low.

<table>
<thead>
<tr>
<th>Metric</th>
<th>File scenario</th>
<th>Collect scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>2.000000</td>
<td>1.433333</td>
</tr>
<tr>
<td>Max</td>
<td>11.666667</td>
<td>8.833333</td>
</tr>
<tr>
<td>Average</td>
<td>4.895652</td>
<td>3.511855</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>2.350787</td>
<td>1.380058</td>
</tr>
</tbody>
</table>

Table 6.3: Average statistics of the measured CPU-usage for 10 Hz

The plots of the raw data in Figure 6.2 shows a very stable behaviour over time, with oscillations between 2% and 11.66667%, which can be assumed for a stable workload.
Collecting data at 100 Hz naturally requires more resources than for 1 and 10 Hz. However, from the results in Table 6.4 we can see that the CPU-usage is quite low with 14.88% for the file scenario and 13.1% for the collect scenario. The majority of the CPU is idle and collecting data at 100 samples per second should not influence the experience of the user.

<table>
<thead>
<tr>
<th>Metric</th>
<th>File scenario</th>
<th>Collect scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>1.633333</td>
<td>1.066667</td>
</tr>
<tr>
<td>Max</td>
<td>24.033333</td>
<td>19.133333</td>
</tr>
<tr>
<td>Average</td>
<td>14.880141</td>
<td>13.10219</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>25.059418</td>
<td>11.299489</td>
</tr>
</tbody>
</table>

Table 6.4: Average statistics of the measured CPU-usage for 100 Hz

The plots of the raw data show larger oscillation than the plots for sampling frequency 1 and 10. However, the CPU-usage does not have any large peaks. The highest peak, and maximum value, for the scenarios is 24% (for the file scenario), which is still low.
1000 Hz

Collecting 1000 samples per second uses nearly half the CPU of the Asus Nexus 7. It involves reading 1000 data packets from the bluetooth input stream, extracting the data values from all of the data packets, 1000 interprocess calls each second, and writing all of the collected data values to a file. The average CPU-usage for the three runs is 45.7% for the file scenario, while the collect scenario used 31.15% of the CPU in average. The majority of the total CPU-usage of the mobile device resides from the BITalino sensor wrapper.

<table>
<thead>
<tr>
<th>Metric</th>
<th>File scenario</th>
<th>Collect scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>40.166667</td>
<td>22.166667</td>
</tr>
<tr>
<td>Max</td>
<td>50.366667</td>
<td>39.833333</td>
</tr>
<tr>
<td>Average</td>
<td>45.700978</td>
<td>31.155003</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>2.956825</td>
<td>8.029407</td>
</tr>
</tbody>
</table>

Table 6.5: Average statistics of the measured CPU-usage for 1000 Hz

The plots of the raw data shows a stable CPU-usage below 50% for both of the scenarios for the whole duration of the experiments, the file scenario had an average CPU-usage of 45.7% and the collect scenario had an average of 31.15%.

![Figure 6.4: CPU-usage for all runs with a sampling frequency of 1000 Hz](image)

(a) CPU-usage of file scenario  (b) CPU-usage of collect scenario

6.1.5 Conclusion

By comparing the average CPU-usage, we can see a significantly increase with higher sampling frequencies, this is due to heavier use of the bluetooth driver and a larger data flow through the system. The CPU-usage is below 50% for all of the sampling frequencies, leaving the majority of the CPU processing power idle.
In Figure 6.5 the CPU-usage of the file scenario is shown in blue, and the collect scenario in red. For all of the sampling frequencies the majority of the used CPU is from the sensor wrapper application, and the overhead introduced by interprocess communication is acceptably low. The CPU-usage when collecting data is so low that it leaves enough resources to perform further computations on the collected data on the phone itself, e.g. visualization or real-time analysis.

6.2 Overnight experiment

This experiment is conducted to demonstrate that the system is suited for collecting data over a longer period where the device is stationary, such as sleep monitoring during a whole night. Additionally, the experiment serves as an evaluation of the energy consumption of both the prototype applications and the BITalino sensor board. A BITalino sensor board is used in the experiment to collect data, which is sent to an Android mobile device that stores the collected data to a file, as illustrated in Figure 6.6.
6.2.1 Description

The scenario is constructed to resemble sleep monitoring using BITalino, with the difference that the various sensors are lying on a table instead of being attached to a person. A BITalino plugged kit with six sensors connected is used to collect data during the experiment. The data values are sent to the mobile device using a bluetooth connection, which the BITalino sensor wrappers (described in the design and impl. chapter) creates and manages. The BITalino sensor wrapper creates JSON objects of the collected data and sends the created objects to the Collector application using interprocess communication defined with AIDL. The Collector application writes the received data to a predefined text file in the external storage of the device. Then, the file is parsed and checked for packet reordering by comparing the timestamp of the received data packets, as shown in Listing 6.2.

```python
with open(sys.argv[1], 'r') as f:
    for line in f:
        data = json.loads(line)
        if data['type'] != "data":
            continue
        count_lines+=1
        print(data['time'])
        cur = datetime.strptime(data['time'],"%H:%M:%S:%f")
        if cur.time() < prev.time():
            count+=1
            prev = cur
        print count
```
Listing 6.3: Python script for detecting packet reordering

The duration of each run is intended to approximate the duration which an average person sleeps during a night. To resemble this a run for each setup was initially conducted for at least eight hours, or until the battery of either the BITalino or the mobile device become empty. This was done to show that the system is able to collect data during a whole night without data loss. To validate that the energy consumption found in the initial runs is correct, three validation runs of two hours each was conducted for each sampling frequency. To get as accurate results as possible, both the devices was charged fully and restarted before starting each run of the experiment.

6.2.2 Performance metrics

For this experiment energy consumption and the actual sampling frequency are used as the metrics to evaluate and analyse the performance of the system. The actual sampling frequency is calculated to see if it is as expected from the BITalino device.

Energy consumption

Energy consumption is measured in milliampere hour, and is defined by how many milliampere hours the mobile device used on average during the experiment.

\[
\text{Battery consumption} = \frac{(\text{start} \% - \text{end} \%) \times \text{battery capacity in mAh}}{\text{duration in hours}}
\]

To obtain the energy consumption the battery percentage when starting and ending each run is written down, and the start time and end time is obtained from the timestamps in the data packets. The battery percentage is given as an integer between 0 and 100, resulting in a rough measure of the used battery. By multiplying the difference in battery percentage with the capacity of the battery and dividing the result by the duration of the experiment, the energy consumption per hour is calculated.

Actual sampling frequency

To check if the actual sampling frequency is as the expected sampling frequency from the BITalino device, the number of received packets are compared with the duration of the experiment. When collecting data from the BITalino device at 1000 samples per second, it is expected to receive 1000 packets each second. By counting the number of received data packets and dividing it by the duration of the run, the actual sampling frequency is calculated. The number of data packets can be obtained by counting the number of lines in the stored file, for instance
by using the `wc -l <file name>` in a Linux based system. The
duration of the experiment can be obtained by calculating the difference
between the timestamps in the first and last data packet in the stored
file.

### 6.2.3 Experiment setup

For the overnight experiment, runs for each of the four sampling
frequencies are conducted. To collect data the BITalino sensor wrapper
and Collector provider application which stores the collected data to a
file, as described in the design and implementation, were used. To
measure the energy consumption a charger is not connected to the
phone, making it rely entirely on the battery, and the screen is turned
off. Initially, a long run for each of the sampling frequencies, is
performed to show that the implemented applications are able to collect
data over a long period. The energy consumption found in the long
initial runs is then validated by performing multiple runs for each of the
sampling frequencies. Because of time constraints the validation runs
are not as long as the initial runs, and are only intended to convince the
reader that the results found in the long runs are correct.

### 6.2.4 Results

In this section, the results for the setups are presented and discussed.
First, the results from the initial runs are presented. Then, the energy
consumption of the shorter runs are presented. No packet reordering
was detected in any of the runs for any of the sampling frequencies.

<table>
<thead>
<tr>
<th></th>
<th>1 Hz</th>
<th>10 Hz</th>
<th>100 Hz</th>
<th>1000 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration (seconds)</td>
<td>35 431</td>
<td>68 511</td>
<td>56 888</td>
<td>27 534</td>
</tr>
<tr>
<td>Data packets</td>
<td>35 431</td>
<td>685 137</td>
<td>5 688 886</td>
<td>27 534 869</td>
</tr>
<tr>
<td>Reordered packets</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>File size</td>
<td>6.0953 MB</td>
<td>117,765 MB</td>
<td>977,86576 MB</td>
<td>4,7605 GB</td>
</tr>
<tr>
<td>Start battery</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>End battery</td>
<td>89%</td>
<td>76%</td>
<td>54%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 6.6: Results from the long initial runs

**1 Hz**

The long initial run for 1 Hz lasted for 9.84 hours (35431 seconds)
before it was stopped. During that time the battery dropped 11%,
corresponding to an energy consumption of 48.3390251693 mAh per hour.

\[
\frac{0.11 \times 4325}{9.84194444} = 48.3390251693 \text{ mAh}
\]
With such an energy consumption, the mobile device can collect data for over 90 hours before it runs out of energy. During the acquisition 35431 data packets were received, which corresponds to one data packet each second as expected.

**10 Hz**

The long run for 10 Hz lasted for 19.03 hours before it was stopped. During that time the battery dropped 24%, corresponding to an energy consumption of 54.54 mAh per hour.

\[
\frac{0.24 \times 4325}{19.0308333} = 54.5430661725 \text{ mAh}
\]

With a energy consumption of 54.54 mAh, the Nexus 7 could collect data for almost 80 hours before running out of energy, assuming that the energy consumption is uniformly distributed. 685137 data packets were received during the 68511 seconds the run lasted. Based on this the number of samples per second was 10.00039, which is very close to the expected 10 samples per second.

**100 Hz**

When collecting data at 100Hz the BITalino ran out of energy after 15.8 hours. At this point the battery of the mobile device had dropped 46%, resulting in an energy consumption of 125.9 mAh per hour.

\[
\frac{0.46 \times 4325}{15.8022222} = 125.90001424 \text{ mAh}
\]

By connecting a larger battery to the BITalino the collection period could be extended. During the 56888 seconds the run lasted, 568886 data packets were received. This corresponds to a sampling frequency of 100.0015 samples per second.

**1000Hz**

After 7.658 hours of data collection at 1000 Hz the battery of the mobile device was empty, resulting in an energy consumption of 565 mAh per hour.

\[
\frac{4325}{7.64833333} = 565.482676211 \text{ mAh}
\]

In order to extend the duration of the data collection a charger needs to be connected to the mobile device. During the run 1000.0315 samples per second were received. The data file containing the received data was approximately 4.7 GB when the data collection ended, which proves that the file size for Android’s file system can exceed 4 GB.
Validation runs

To validate the energy consumption found in the initial long runs, three shorter runs for each sampling frequency were conducted. For the short runs the setup was identical to the long runs, but the duration of each run was only three hours. The energy consumptions found in these runs are not as accurate as the long runs, since the current energy of the battery is given as an integer percentage. If the battery has a total capacity of 4325 mAh, the energy consumption is given in steps of 43.25 mAh, where each step corresponds to one percentage (i.e. 5% battery power used means that \( 5 \times 43.25 = 217.5 \) mAh was used) To compensate for this inaccuracy the energy consumption is given as a interval between two of these steps; the step corresponding to the percentage currently displayed, and the step corresponding to the percentage below. Once the interval is determined, it is divided by 3 hours to indicate the energy consumption per hour.

<table>
<thead>
<tr>
<th>Sampling frequency</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Hz</td>
<td>43.25 - 57.666</td>
<td>43.25 - 57.666</td>
<td>43.25 - 57.666</td>
</tr>
<tr>
<td>10 Hz</td>
<td>43.25 - 57.666</td>
<td>57.666 - 72.0888</td>
<td>43.25 - 57.666</td>
</tr>
<tr>
<td>100 Hz</td>
<td>115.333 - 129.75</td>
<td>115.333 - 129.75</td>
<td>115.333 - 129.75</td>
</tr>
<tr>
<td>1000 Hz</td>
<td>562.25 - 576.666</td>
<td>547.833 - 562.25</td>
<td>562.24 - 576.666</td>
</tr>
</tbody>
</table>

Table 6.7: Results from the short validation runs

The energy consumption of the short validation runs is approximately the same as for the long runs. Take the first validation run with a sampling frequency of 1 Hz as an example; When the three hour long run ended the energy consumption was between 3% and 4% of the Nexus’ battery, which can store a total of 4325 mAh. This correspond to a energy consumption between 43.25 mAh and 57.666 mAh. The initial run for 1 Hz used 48.339 mAh per hour, which is within the interval. The results for the validation runs, presented in Table 6.7, implies that the results from the initial runs are representative.

6.2.5 Conclusion

The results of the experiment show that the functional requirements for the system are satisfied, the system managed to collect data during a whole night even for a high sampling frequency. Moreover, the system is able collect data and it is stable and robust enough to do it for a long period. The energy consumption increased significantly for higher sampling frequencies. The energy consumption with a sampling frequency of 1000 Hz revealed that the mobile device potentially could run out of power. This can be solved by connecting a charger to the mobile device when collecting data at a high sampling frequency.
The measured sampling frequency of the BITalino is very close to the configured sampling frequency. One possible reason for the minor deviation from the expected sampling frequency is that the difference between start and end time is not calculated with high enough precision.

6.3 Bluetooth stability experiment

One of the intended usages of the system is sleep monitoring at home using the BITalino sensor board. To be able to receive the data collected by the BITalino the mobile device needs to be within bluetooth range of the BITalino. During the night the monitored person might need to use the toilet, or leave the bedroom for other reasons. However, if the toilet is outside the bluetooth range the connection is lost. To cope with this problem a reconnection mechanism (as described in the Chapter 5) is implemented.

To test this reconnection mechanism a series of simple experiments are conducted. The goal of the experiments is to demonstrate that the reconnection mechanism works and to determine whether factors like the length of the disconnection and the number of disconnections could have any impact. As such the experiment is designed rather simple and does not put major effort into high accuracy of the measurements. To achieve higher accuracy is necessary to know exactly when the devices are within range of each other.
The results presented in this section are not precise, and are just intended to convince the reader that the reconnection mechanism works as intended.

6.3.1 Experiment setup

To evaluate the reconnection mechanism a set of experiments were conducted. Each experiment was designed to simulate a scenario where the monitored person leaves the bluetooth range for a period of time. Experiments were conducted using a BITalino sensor board. When the data acquisition had started the BITalino board was moved outside the bluetooth range for a period of time. After the given period of time the BITalino was moved back within the bluetooth range and the reconnection time was measured. To determine if the devices were connected the BITalino's LED light was used. As stated in the background chapter the LED blinks at different frequencies depending on the state of the BITalino. The devices are considered connected when the BITalino is collecting data, indicated by the LED blinking at 1 Hz.

6.3.2 Performance metrics

In this experiment, the reconnection delay, measured in seconds, is used as the metric to evaluate the bluetooth reconnection mechanism. To measure the time used to reconnect a stopwatch was started when the devices are approximately within range of each other, and stopped when the LED started blinking faster.

Reconnection delay

The reconnection delay is defined as the number of seconds used to reconnect when the devices are within bluetooth range of each other.

6.3.3 Tests

The reconnection mechanism is evaluated by two tests, one test where the disconnection time varied, and one test where the devices are disconnected and reconnected multiple times. This was done to investigate how time and the number of re-connections influenced the reconnection mechanism.

Time test

In this test, the devices were moved out of bluetooth range of each other for various amounts of time, before they are moved within range of each other again. This were done to investigate if there is any differences if the devices was apart for a short period of time or a long period of time. The devices were outside the range of each other for:

- less than 30 seconds,
• between 1 and 2 minutes,
• between 2 and 5 minutes,
• more than 1 hour.

The BITalino device was turned off and on between each experiment.

**Repetition test**

In this test, the devices were moved outside each other bluetooth range, and then immediately moved within range of each other again. This procedure was repeated several times to investigate the influence of the number of reconnections without turning the device on and off.

### 6.3.4 Results

In this section the results found during the experiments are presented and discussed. The results for the time test is presented in Table 6.8.

<table>
<thead>
<tr>
<th>Time outside range</th>
<th>Reconnection delay (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 30 seconds</td>
<td>2, 3, 4, 3, 1</td>
</tr>
<tr>
<td>Between 1 and 2 minutes</td>
<td>10, 5, 15, 10, 22</td>
</tr>
<tr>
<td>Between 2 and 5 minutes</td>
<td>22, 3, 10, 15, 25</td>
</tr>
<tr>
<td>More than 1 hour</td>
<td>16, 28, 10</td>
</tr>
<tr>
<td>More than 5 hours</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table 6.8: Results of time test**

For the time test we can see that the reconnection is successful in all of the runs. The reconnection delay is less than the largest sleep period in all of the runs, which implies that the reconnection was successful at the first attempt after re-entering the bluetooth range of the devices. The reconnection mechanism does not seem to be affected by the time spent disconnected. In Table 6.9 the results for the repetition scenario is presented.

<table>
<thead>
<tr>
<th>Run number</th>
<th>Disconnections</th>
<th>Reconnections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

**Table 6.9: Results of repetition test**

The results from the repetition test show that the reconnection was successful for all of the disconnections in each run. This indicates that the reconnection mechanism is not affected by the number of disconnections in a sleep monitoring scenario. At least not if the person leaves the bluetooth range of mobile device less than 20 times.
The reconnection mechanism was also tested by turning off the BITalino device, to approximate a scenario where either the monitored person accidentally turns off the BITalino before turning it on again, or the BITalino runs out of power and is recharged. The results for the runs conducted in this scenario showed approximately the same results as for the time test; the reconnection mechanism was able to reconnect in all of the runs.

6.3.5 Conclusion

The reconnection was successful in all the runs in the experiment. Since the longest sleep period (the period before the smart phone attempts to reconnect) is 30 seconds, the worst case reconnection delay should be \(30 + a\) seconds, where \(a\) is the time period used by a reconnection attempt, assuming that the reconnection is successful by the first reconnection attempt after re entering the range. From the results we can see that the longest reconnection delay was less than \(30 + a\) seconds, which implies that the reconnection was successful at the first reconnection attempt in all of the runs.

The time of the reconnection attempts can be calculated with the following formula. Where \(R_N\) is reconnection attempt number \(N\), and \(a\) time of a reconnection attempt.

\[
R_N = \sum_{n=0}^{N} \begin{cases} 1 \times 2^n, & \text{if } n < 5 \\ 30, & \text{if } n \geq 5 \end{cases} + N \times a
\]

Assuming that the time of each reconnection attempt, \(a\), is 5 seconds the mobile device will try to reconnect after 1, 8, 17, 30, 51, 86, and 121 seconds.

The BITalino sensor wrapper is designed to collect data while the monitored person is asleep. With a reconnection delay of less than \(30 + a\) seconds the amount of lost data readings from each sensor is at most \((30 + a) \times f_s\), where \(f_s\) is the sampling frequency. The amount of lost data is negligible, since very few people falls asleep within \(30 + a\) seconds after entering bluetooth range of their bed, assuming that the mobile device is next to the bed.

6.4 Summary of results

In this section, the key results from the experiment chapter are presented and discussed. Three experiments were conducted as a part of the evaluation; the CPU usage of the mobile device was measured when collecting data from a BITalino sensor board, the energy consumption was measured when collecting data from a BITalino sensor board during a whole night, and, lastly, the bluetooth mechanism of the BITalino
sensor wrapper was tested. All of the experiments were conducted using a Asus Nexus 7 and a BITalino plugged kit.

The CPU-usage was measured when collecting data, and storing the collected data to a file, at four different sampling frequencies (1, 10, 100 and 1000 Hz), from a BITalino sensor board with 6 sensors attached. The experiments showed that the CPU-usage increased significantly when the sampling frequency was increased, as shown in Figure 6.8.

![Figure 6.8: Comparison of the average CPU-usage of the different sampling frequencies](image)

For all of the four sampling frequencies the average CPU-usage was below 50% of the total CPU capacity of the mobile device. Moreover, for all of the sampling frequencies except 1000 Hz the average CPU-usage was below 20%. With the majority of the CPU capacity unused, even when collecting 1000 samples each second, it is feasible to perform real-time analysis of the collected data simultaneously. Performing real-time analysis on the mobile device will decrease the amount of network traffic significantly compared to sending all of the collected data to a centralised computer for analysis. In such a scenario the centralised computer could receive a message if a certain event occurred; for instance if an elderly has fallen and has not moved for a period.

Of course, higher CPU-usage impacts the energy consumption of the data acquisition system. From the battery usage found in the overnight experiment there is a clear correlation between the energy consumption and the CPU-usage for different sampling frequencies.
As for the CPU-usage, the energy consumption was significantly higher when collecting data at 1000 Hz. Assuming a uniform energy consumption, it is possible to collect data at 1000 Hz, using a BITalino sensor board, for 7.65 hours based on the results from the overnight experiment. The fact that it was possible to collect data through a whole night, and even longer, shows that the functional requirements are met. The BITalino sensor wrapper is modelled to monitor sleep on persons which might have Obstructive sleep apnea. During a night of sleep the monitored person might leave the bedroom, and the phone, to visit the toilet, which might cause a bluetooth disconnection between the mobile device and the BITalino device. To cope with this a reconnection mechanism was implemented, and some simple experiments were conducted to validate that it worked. In all of the experiments the devices reconnected within $30 + a$, where $a$ is the time a reconnection attempt. Resulting in a minimal data loss in a sleep monitoring scenario, since few people fall asleep within 30 seconds after entering bluetooth range of the phone, and the bed. Only the data collected while the monitored person is asleep is relevant when determine if a person has Obstructive sleep apnea, so one could argue that the probability of losing relevant data is very small.
Part IV

Conclusion
Chapter 7

Conclusion

7.1 Summary

Modern mobile devices become increasingly powerful, and are capable of collecting data from a large range of both built-in and external sensors. These sensors might use different communication channels and data exchange protocols, making it burdensome to adopt new data sources for collecting data; sensor-specific knowledge (such as the Link Layer technology and communication protocol) is required to implement the support of a data source. This implementation is often performed once for each application that uses the collected data. Implementing the support for each sensor once and reusing the code across applications would reduce the amount of duplicate code and development. Additionally, some applications might want to exchange a sensor with another sensor (the application developer might realise that the data quality of a sensor is not satisfactory) without changing the rest of the application. In order to do this they would have to implement the support for the new sensor and integrate this with the rest of the application.

To address the above-mentioned issues a push-based data acquisition system for Android is proposed in this thesis. The design makes as few assumptions as possible to support all current and future sensors. The system separates the software into two components, a sensor wrapper and a provider. A sensor wrapper is created for each data source supported by the system to hide the sensor specific details, and is responsible for establishing a connection and for collecting data from the data source. Any application is free to use these sensor wrappers to collect data from a data source. The applications that make use of these sensor wrappers are called provider applications. A provider application is responsible for starting and stopping the data acquisition using the sensor wrappers, and for using the collected data (in any way desired by the application developer). The provider application can start and stop the data acquisition through a common interface for all sensor wrappers. By exposing all sensor wrappers through one common interface, the developer of the provider applications only needs to know the
details of one interface instead of the details of all used data sources. The interface exposes a command for discovering the sensor wrappers that are available on the mobile device, by performing a handshake with the sensor wrappers to obtain the ID of those sensor wrappers that are available. The IDs are later used to specify which sensor wrappers to start when starting the data acquisition, enabling the possibility to acquire data from any subset of the available sensor wrappers. When a sensor wrapper receives a signal to start data acquisition it establishes a connection to the data source and starts collecting data from it. The data is then expressed in a well-defined data format as a JSON object and sent to the provider application that started the data acquisition using interprocess communication as soon as a new value is available. The sensor wrapper and the data format preserves the context of the data acquisition by sending the details of the context (such as what each sensor measured, the unit it is measured in and an optional description) as metadata to the provider application. Additionally, there are certain other requirements to such a data acquisition system. For one thing, the system needs to be resilient with respect to stability, data loss, packet reordering, security and other unforeseen events to provide expected behaviour to the user. Furthermore, the resource usage of the system should be moderately low, and leave enough resources to perform further computations with the collected data.

As a part of this thesis a prototype sensor wrapper application and provider application was created to evaluate the design and to expedite the CESAR project. The CESAR project aims at enabling anyone to perform sleep monitoring and analysis at home at a low-cost using a mobile device by initially collecting data using a BITalino sensor board. The first step towards achieving this is to be able to collect data from the BITalino sensor board and to store the collected data. The prototype sensor wrapper is tailored to collect data from a BITalino sensor board, which the provider application either stores to a file or forwards using TCP. The BITalino sensor wrapper is implemented using the Java SDK provided by BITalino. During the development of the sensor wrapper some bugs and shortcomings were discovered in the SDK. When discovered they were fixed and they are now merged with the SDK.

By design, and by implementing prototype applications, the extensibility of the data acquisition is shown. To show that the rest of the requirements are met, and to show that the prototype is suited for sleep monitoring, three different experiments were conducted. In all of the three experiments an Asus Nexus 7 was used together with a BITalino sensor board with a sensor connected to each of the six analog inputs. The sampling rate that data was sampled at from the connected sensors varied between the experiments. First, data was collected during a whole night to determine the stability and energy consumption of the implemented prototype. The collected data was stored to a file, which was later parsed to detect data loss and packet reordering.
No data loss or packet reordering was detected during the experiment, and the prototype was able to collect data during the whole night in all of the runs. This indicates that the prototype is stable, and therefore it is well suited for performing long running data acquisition (such as sleep monitoring). Next, the CPU-usage of prototype applications was measured while collecting data from the BITalino sensor board at four different sampling frequencies (1, 10, 100 and 1000 Hz). The experiments showed that the CPU-usage increased significantly when the sampling frequency was increased. On average the CPU-usage for all four sampling frequencies was below 50% of the total CPU capacity of the mobile device. However, for all sampling frequencies except 1000 Hz the average CPU-usage was below 20%. With the majority of the CPU unused, even when collecting data at 1000 Hz, it is feasible to perform further computations (such as real-time analysis) on collected data on the mobile device itself. Lastly, the Bluetooth reconnection mechanism implemented in the BITalino sensor wrapper was evaluated. During a night of sleep the monitored person might leave the bedroom and the phone, to visit the toilet. This might cause a bluetooth disconnection between the mobile device and the BITalino device. To cope with this a reconnection mechanism were implemented, and some simple experiments were conducted to investigate the effects of disconnection time and the number of disconnections. In all of the runs the devices successfully reconnected with each other within $30 + a$ seconds, where $a$ is the time of a single reconnection attempt. This approach leads to an acceptable amount of data loss in a sleep monitoring scenario, since few people fall asleep within 30 seconds after entering bluetooth range of the phone, and the bed. Only the data collected while the monitored person is asleep is relevant when determine if a person has OSA, so one could argue that the probability of losing relevant data is very low.

As mentioned above, the implemented provider application either stores the collected data to a file or forwards it using TCP, which creates multiple use cases for the prototype. The possibility of storing the collected data to file enables data acquisition, and especially sleep monitoring, at locations without a stable internet connections without data loss. Moreover, it can be used by the CESAR project to monitor individuals that do not have an internet connection. The next day, the collected data can for instance be analysed using data mining algorithms to determine if the monitored individual has OSA. If a stable internet connection is present the collected data can be forwarded to a computer using TCP instead, which ensures that all of the sent data arrives at the receiving end. Once received at the receiving end the possibilities are numerous. The data could for instance be analysed real-time using Complex Event Processing, be visualized in plots, or simply be stored to a file or a database. To demonstrate some of the possibilities of the latter approach multiple Python scripts were developed (see Appendix D). One of the scripts visualizes the received data real-time by plotting it using Kst, one stores the collected data to a SQLite database. The
last script uses a simplification of Complex Event Processing to analyse accelerometer values collected by the mobile device. Based on the received values it determines if the mobile device has fallen, if so the user is alerted by a message printed to the terminal.

In [50], Brunette et. al. proposed an extensible data acquisition framework for Android called Open Data Kits Sensors (ODK Sensors). We were not aware of their framework until after the design and implementation of our prototype was completed. Therefore, we use their framework to evaluate the design decisions made in this thesis. The focus of ODK Sensors is to enable data collection from any kind of sensor (wired and wireless), and to simplify application development by creating one interface that can control virtually any kind of sensor. ODK Sensors is a part of the larger Open Data Kit [51] project that “seeks to develop a modular set of tools to magnify human resources through appropriately designed technology” [50]. As a result of this, the ODK Sensor framework can only be used as a part of an ODK project. Multiple designs of the framework were proposed, one of which proposes to create a data acquisition framework which uses software drivers to implement the sensor specific details (such as connection and communication). The sensor framework multiplexes the data collected by the various sensor drivers through a ContentProvider, which other applications can perform queries against in order to obtain the collected data. However, this introduces two steps of IPC before the data is received by the application which requested it. It also introduces conflicts in resource management and ownership, and it is challenging to decide how long and to whom certain data should be available. Three versions of the framework with different IPC mechanisms were implemented and evaluated with respect to flexibility and performance. After testing the three framework implementation they concluded that the software separation with separate driver applications (i.e. used in this thesis) offered the best tradeoff in terms of programming ease, deployment ease, and performance. There are many common aspects of the ODK Sensor framework and the data acquisition system developed in this thesis. Both reduce duplicate code and enable data acquisition from current and future sensor with an easily extendible system. The ODK Sensor framework can only be used together with the larger ODK project framework, which might be unnecessarily extensive and undesired for certain applications. Unlike the ODK Sensor framework, this thesis proposes an independent and lightweight data acquisition system that can be used by any application.

The proposed design satisfies the requirements posed in Section 4.1. The various experiments show that the implemented prototype provides the desired resilience and resource usage. Furthermore, the overnight experiments show that the prototype can handle a high workload without saturating. The developed prototype is well suited for monitoring sleep since it is able to collect data over a long period of time.
without data loss or packet reordering, and the mobile and BiTalino device reconnect within reasonable time when a disconnection occurs. Collecting medical data raises privacy concerns, especially when dealing with sensitive health information. Loss and/or modification of medical data pose financial risks to firms and medical risk to patients. Data acquisition is often used to determine if the treatment of a patient is working as intended, and modification of the collected data could lead to wrong diagnosis, wrong medication, and further treatment of the patient; this is potentially life-threatening for the patient in extreme cases. The sensitive information can be exploited in many ways, all of them raising a concern about privacy. However, the privacy concern is not addressed in this thesis; it is only briefly discussed in the future work section.

By design the data acquisition system is easily extensible, both in terms of adding support for a new sensor (one only needs to implement the sensor specific details) and in terms of enhancing the functionality of the mobile device to use already supported sensors (which can be downloaded from an application marketplace). The proposed design makes external sensors more accessible for application developers. They only need to deal with the proposed interface instead of knowing the sensor specific details of every sensor they want to use. In an ideal scenario, the manufacturers of each data source would create the corresponding sensor wrapper. This way they are responsible and able to ensure correct use of the data source. Moreover, the fact that all supported sensors can be used through one interface makes it much easier to change which data source to collect data from. The application only needs to select another sensor wrapper to start data acquisition (by using the ID of the new sensor wrapper instead), instead of implementing the sensor specific details of the new data source. Simplifying this can make a huge difference for application developers, which might discover that the sensor they use does not provide the desired data quality. It increases the chances that the application developer actually changes the data source instead of accepting that the quality is poor. Additionally, by using a well-defined format for representing the collected data the context of the data acquisition is preserved, and it does not matter which data source the data originated from since all sensor wrappers represent the data in the same format.

To recap: the proposed design enables reuse of sensor specific code between applications which decreases the amount of duplicate code and makes sensors more available. The simple deployment method makes external sensors more accessible, and it will hopefully lead to the creation of new applications which are dependable on external sensors. The prototypes developed in this thesis provides stable data acquisition from the BiTalino sensor board, while making it feasible to perform further computations on the collected data by keeping the resource usage low. It should provide a good starting point for the CESAR project.
provided that the data quality of the BITalino is good enough.

7.2 Open problems and future work

The data acquisition system implemented in this thesis is a prototype, and, as such, not a complete product; multiple iterations are necessary to make it complete. In this section, open problems with the prototype are discussed, and suggestions for future work are given.

7.2.1 Open problems

The focus of this thesis is to create an extensible data acquisition system, which stores the collected values to a file or sends it to a server for processing. Therefore, the sensor wrappers create JSON-objects of the collected data. The chosen high-level design enables further processing of the data on the mobile device by the provider application, but this would require the provider application to parse the created JSON-objects to extract the data values, resulting in unnecessary encoding and decoding of the data. In retrospect a better solution would have been to move the creation of JSON-objects from the sensor wrapper applications to the provider applications. By doing this the unnecessary creation and parsing of JSON-objects is avoided when processing the data in the mobile device.

By design, the system is not only limited to collect data from the BITalino sensor kit; it is possible to add support for other data sources as well. In this thesis only support for the BITalino sensor kit is added to the system. However, by creating sensor wrapper applications (adding support) for new data sources the system would become more attractive to use by application developers. All of the biomedical sensors discussed in Chapter 2 are examples of sensors that should be added to the system. Adding support for the e-Health sensor platform (by Cooking Hacks) might be particularly interesting for the researchers of CESAR project, since the sensor platform contains a nasal airflow sensor (which the BITalino sensor kit does not).

In Android 6.0, Doze was introduced. Doze is a power saving mechanism in the Android operating system, which puts applications to sleep if the mobile device is not being used. The device is considered not used if the charger is not plugged in, the screen is turned off, and has been idle for a longer period. When collecting data over a long period (for instance during a whole night) all of the three conditions might be satisfied, and Doze might kick in. Unfortunately, the documentation of Doze is not comprehensive and the behaviour of Doze unknown. Further research should be conducted to determine the behaviour and impact of Doze during data acquisition.
7.2.2 Future work

The implementation of the data acquisition presented in this thesis is only a prototype, and protection of sensitive data should be added before using it on a big scale. In this section, some of the potential security breaches are identified and discussed, focusing on the sleep monitoring scenario with the implemented provider and sensor wrapper application for a BITalino device. In this scenario, physiological sensors are connected to the monitored person, and these sensors are sampled by the BITalino device which sends the data to a mobile device using bluetooth. The values are received by a sensor wrapper application on the mobile device, then sent to another application, using a Binder, which stores the collected data in a file on the external memory partition of the mobile device.

Sensors can be connected to the mobile device over wireless channels, as bluetooth for the BITalino. Wireless connection requires authentication and encryption to provide the desired level of privacy. The BITalino used in the sleep monitoring scenario uses bluetooth 2.0 and a legacy pairing (using a PIN-code) for authentication. The fact that the PIN-code used to establish a connection with the BITalino device is 1234 makes the connection insecure. To increase the security of the wireless connection additional authentication or encryption is necessary. Another possible weakness of the system is how the collected data is stored. The current prototype stores the collected data to the external memory partition of the mobile device without encryption. This partition is shared between applications, and proper encryption is needed to protect the data.

The implemented prototype of the data acquisition system provides a solid starting point for the CESAR project, by enabling reliable data acquisition from a BITalino sensor board over a long period. However, the quality of the data collected by the BITalino sensor board is unknown and needs to be evaluated to ensure reliable results. This can, for example, be done by comparing the collected data from each of the sensors with the gold standard of the given sensor. Furthermore,
multiple sensors of each sensor type should be evaluated to detect diversity. Based on the results of the evaluation, the researchers of the CESAR project can either give a measurement of how accurate the data acquisition is, or they might decide to use another data source to collect the data (such as Cooking Hacks’ E-Health Sensor Shield). Fortunately, such an exchange is facilitated by the proposed design.
Appendices
Appendix A

Creating a new driver application

To ease the creation of a new sensor wrapper application a template project is provided, the creator of the new driver application just need to fill in the blanks. These blanks are marked by numbered TODOs in the code of the template project and the corresponding action for each TODO can be found in this chapter. Creating a new sensor wrapper involves implementing connection, disconnection and data extraction based on the communication channel and protocol of the data source, which is described in section 8.2. The provided template project is used in this manual to describe how to create a new sensor wrapper application. Therefore, a description of how to download and set up the template project is given first.

A.1 Project setup

The first step in creating a new sensor wrapper application, is to download the template project and update it with the names of the new application. This step by step walk-through is based on using Android Studio on a Unix based system.

A.1.1 Download the project

The first thing needed is the template project itself, which is found on GitHub. Download the template project, rename it with the desired name and move it to the Android Studio Project-folder, as shown in Listing A.1.

```
1  git clone git@github.com:sveinpg/template.git
2  mv template <path to Android Studio Project folder>/<new_directory_name>
```

Listing A.1: Download and rename the template project
A.1.2 Import the project to Android Studio

When the project is downloaded and move, the next step is to import it to Android Studio. This is done by launching Android Studio and opening the project, as shown in Listing A.2.

1. File->Open->(Select the downloaded folder)

Listing A.2: Importing template project to Android Studio

A.1.3 Rename the packages

The packages in the template project is named com.sensordroid.templatedriver, which might be undesired in the new project. To change the package name right click the package you want to rename and follow the steps illustrated in Listing A.3.

1. Right click -> Refactor -> Rename -> Rename package -> *type desired name* -> Refactor

Listing A.3: Packet re-factoring

It is important to not change the package name of the AIDL-interface; the system would not recognize the interface as the same as the interface of the provider application, breaking the interprocess communication.

A.1.4 Change the application ID

Each project in android is uniquely identified with by its application ID. This ID is used to separate applications from each other, hence a new sensor wrapper application needs a new application ID. To change this value open the build.gradle file inside the app folder of the project, and change the applicationId to your new ID, see Listing A.4. The location is marked with TODO 1 in the template project.

1. location: app/build.gradle
2. android {
3.   defaultConfig {
4.     //TODO 1: change applicationId to your ID
5.     applicationId "com.sensordroid.templatedriver"
6.   }
7. }

Listing A.4: Updating the ID of the project
A.1.5 Change the name of the application

In addition to an ID each Android application got a name. This name is displayed to the user as the name of the application in the program menu, and is the name used when registering the sensor wrapper to a provider application. This name is specified by the `app_name` variable in the `string.xml` file, which is located in the `app/res/values` folder. The location is marked with TODO 2 in the template project, and give the application a new name the `app_name` variable is given a new value, see Listing A.5.

```xml
<resources>
  <!-- TODO 2: Change "TemplateDriver" to the name of the application-->
  <string name="app_name">TemplateDriver</string>
</resources>
```

Listing A.5: Changing the name of the application

A.2 Adapting to the data source

At this point the (renamed) project should be imported to Android Studio. The next step is to implement the data source specific details. Different data sources might use different communication channels and protocol, adapting a data source includes implementing the connection to the data source, the collection of data, and the disconnection from the data source. To illustrate how to do this a BITalino sensor board is used as an example.

When the data collecting is signalised to start, by the provider application, a separate thread (`CommunicationHandler.java`) is created for communication with the data source. The thread connects to the data source, and sends the collected data to the provider application until it is signalised to stop (by a call to the `interrupt()` method on the thread). The connection, collection and disconnection associated with a data source is implemented in this thread, and variables used across these actions are stored as class variables (to be accessible across actions). An inputstream would for instance fall into this category, since it is initialized when connecting to the data source, read from when collecting data and closed when closing the connection to the data source.
A.2.1 Sending metadata

A.2.2 Send metadata

To preserve the context of the data acquisition metadata about the data source is sent to the provider application. The metadata is data source specific, and must therefore be implemented by the sensor wrapper developer. We propose to implement this as a runnable MetadataHandler object (marked with TODO 10).

Start with obtaining the metadata from the shared preferences and create a metadata packet using the JSONHelper library. Then send the created data packet using the binder object. When reading the values from shared preferences a default value need to be provided in case the key is not present. In the example below the values saved to the shared preferences above is read and sent as a metadata packet.

```
MetadataHandler.java
1 // Collect values from shared preferences
2 SharedPreferences sharedPref =
3     getActivity().getSharedPreferences( "com.example.preferences", Context.MODE_PRIVATE);
4 String default = " ";
5 String type = preferences.getString("type", default);
6 String metr = preferences.getString("metr", default);
7 String desc = preferences.getString("desc", default);
8 // Create metadata packet
9 String sendString = JSONHelper.metadata(name, id,
10       new int[]{0}, new String[]{type},
11       new String[]{metr}, new String[]{desc}).toString();
12 binder.putJson(sendString);
```

Listing A.6: Creating and sending metadata

For more information about how to use the JSONHelper to build data packets, see the section about building data packets.

A.2.3 Connecting to the data source

In order to collect data from a data source a connection to it is required. The connection to the data source is implemented in the connect() method (marked with TODO 3) in the communication thread. This implementation is required when creating a new sensor wrapper application, since the Link Layer technology used by a data source might vary. In Listing A.7, an example creating a connection to a BITalino using bluetooth is presented. The BITalinoDevice, which contains the output- and input-stream of the Bluetooth socket is stored as a class variable.
public void connect() throws IOException, BITalinoException {
    final BluetoothAdapter blueAdapt =
        BluetoothAdapter.getDefaultAdapter();
    final BluetoothDevice dev =
        blueAdapt.getRemoteDevice(remoteDevice);
    final List<UUID> uuidList = new ArrayList<>();
    ParcelUuid[] uuidParcel = dev.getUuids();
    boolean connected = false;
    for (ParcelUuid uuid : uuidParcel) {
        BluetoothSocket tmp;
        try {
            tmp =
                dev.createInsecureRfcommSocketToServiceRecord(uuid.getUuid());
        } catch (IOException ioe) {
            ioe.printStackTrace();
            continue;
        }
        mSocket = tmp;
    }

    blueAdapt.cancelDiscovery();
    try {
        mSocket.connect();
        connected = true;
        break;
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}

if (!connected) {
    throw new IOException("Could not connect to bluetooth device");
}

bitalino = new BITalinoDevice(SAMPLING_FREQ, ids);
bitalino.open(mSocket.getInputStream(),
    mSocket.getOutputStream());
}

Listing A.7: Establishing a connection to a BITalino sensor board
In the code the UUIDs of the paired Bluetooth devices is collected. The UUIDs are used one by one to try and connect to the bluetooth device until a connection is successfully established. If the connection is successful a BITalino is created and opened, which stores the input- and output-streams of the bluetooth socket. These streams are later used to collect data from the data source.
A.2.4 Collecting data from the data source

Once a connection is established the `collectData()` method is called, which collects data from the connected data source until the thread is interrupted. The process of collecting data might vary depending on the data source. If for instance the data source is a text-file the data collection could be to periodically read a line from the file. In a more complex scenario, as when using a BITalino device, the data collection involves sending a command to the BITalino device to start the and then read data packets from the bluetooth input-stream. When the data is collected from the data source it is passed to a worker thread as a runnable `DataHandler` object, for further processing as described in section A.8.

```java
app/src/main/java/com/sensordroid/templatedriver/Handlers/CommunicationHandler.java
*/

private void collectData() throws IOException,
BITalinoException {
  bitalino.start();
  while (!interrupted) {
    if (Thread.currentThread().isInterrupted()){
      interrupted = true;
      break;
    }
    final BITalinoFrame[] frames;
    try {
      frames = bitalino.read(FRAMES_TO_READ);
      for (final BITalinoFrame frame : frames) {
        executor.submit(new DataHandler(binder, frame,
                                          driverId));
      }
    } catch (BITalinoException e) {
      e.printStackTrace();
    }
  }
}
```

Listing A.8: Collecting data from a BITalino device

For a scenario using a BITalino device as the data source, the data collection involves calling `bitalino.read(int n)`, which reads `n` number of data packets from the input-stream and creates `BITalinoFrame` objects of these. A `BITalinoFrame` (hereafter known as a frame) is an object containing the values of the received data packet, like the collected data values. Each of the frames are passed to a worker thread which is responsible for creating JSON-objects and sending it to the main application.
A.2.5 DataHandler

The runnable DataHandler is responsible for preprocessing the collected data, before sending it to the Provider application. A DataHandler object (containing the collected data) is created in the collectData() method to pass the collected data to a worker thread. The type of the object containing the collected data (as the BITalinoFrame for a BITalino device) might vary between data sources. Therefore, the constructor of the DataHandler object needs to be changed accordingly. In Listing A.9, the constructor is changed to take BITalinoFrame as an argument, and the type of the class variable, data, used to store the data is changed accordingly.

```java
public class DataHandler implements Runnable {
    private static IMainServiceConnection binder;
    private static BITalinoFrame[] data; // <-- Changed from BITalinoFrame[]
    private static int id;

    // TODO 7: Change type of "data" to match your format
    public DataHandler(IMainServiceConnection binder,
                        BITalinoFrame[] data, int id) {
        this.binder = binder;
        this.id = id;
        this.data = data;
    }
}
```

Listing A.9: Changing the constructor of the data-object

The next step is to rewrite the data passed to the DataHandler before sending it over the binder object. In this context rewriting the data means extracting the sampled values from the data-object and creating a list of the extracted values. When the runnable DataHandler is run a JSON-object is created of the collected data, before it is sent to the Provider application with a remote method call (putJson(String)). The method call is made locally to the binder variable, and executed remotely in the Provider application. A helper function (JSONHelper.construct()) is provided to ensure that the data packets are created correctly. The helper function takes three arguments; the ID (id) of the sensor wrapper, the a list of ints containing the IDs of the data channels (determined by the sensor wrapper developer), and lastly a list of the data corresponding to the IDs (the extracted values from the data object).

As mentioned earlier, the data object used to pass the collected data from a BITalino device is a BITalinoFrame. Extracting values from the BITalinoFrame can be done by calling the getAnalog(int pos) method, as in Listing A.10.
// TODO 6: Extract values from data.
final int[] channels = new int[]{0,1,2,3,4,5};
Object[] values = new Object[]{data.getAnalog(0), data.getAnalog(1), data.getAnalog(2), data.getAnalog(3), data.getAnalog(4), data.getAnalog(5)};

String sendString = JSONHelper.construct(id, channels, values).toString();
binder.putJson(sendString);

Listing A.10: Extracting and sending the collected data
An object array is created and passed to the JSONHelper which constructs a data packet of the given arguments.

A.2.6 Closing connection

When a interrupt() call is done to the communication thread the data collection is stopped, and the connection to the data source is closed and cleaned up. The closing and cleaning up is implemented in the resetConnections() method, which closes all the connections opened in the connect() method. If for instance a socket was opened, the socket should be closed. In the BITalino example we connected to a bluetooth socket and created a BitalinoDevice. In the resetConnection() method, they are both closed, see Listing A.11.

private void resetConnection(){
    Log.d("debug", "entering resetconnection");
    if(bitalino != null){
        try {
            bitalino.stop();
            bitalino = null;
            Log.d(TAG, "Bitalino is stopped");
        } catch (BITalinoException e) {
            e.printStackTrace();
        }
    }
    if (mSocket.isConnected()){
        try {
            mSocket.close();
            mSocket = null;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

Listing A.11: Closing the connections to a BITalino sensor board
A.3 Configuration

Different sensor wrapper applications got different needs in terms of configuration. In this section an example setting up the metadata of one channel will be given. This includes saving the data type, metric and description and sending the saved data as metadata.

A.3.1 Defining the layout

Create the desired layout to save your configuration. In our example three values should be saved, so three EditText elements is added to the layout file. As well as a button used to trigger the saving of the entered values.

```xml
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/editType"
    android:hint="Data type" />
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/editMetric"
    android:layout_below="@+id/editType"
    android:hint="Metric" />
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/editDescription"
    android:layout_below="@+id/editMetric"
    android:hint="Description" />
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Save configuration"
    android:id="@+id/button"
    android:layout_below="@+id/editDescription"/>
```

Listing A.12: Layout definition of sensor wrapper

A.3.2 Storing the configuration

Add listeners according to the layout you created, and save the entered configuration to shared preferences. The shared preferences can be accessed from anywhere in the application and can even be accessed after a relaunch of the application. For more information about writing and reading from shared preferences see the chapter about shared preferences.
In our example we add a listener to the button from our layout. When the button is clicked, the text from the EditText fields are collected and written to shared preferences.

```java
public void onCreate(Bundle savedInstanceState) {
    // Obtain shared preferences object
    SharedPreferences preferences =
            getSharedPreferences("com.example.preferences", Context.MODE_PRIVATE);
    Button saveButton = (Button)findViewById(R.id.button);
    saveButton.setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View view) {
            // Get EditText-objects
            EditText type =
                    (EditText)findViewById(R.id.editType);
            EditText metr =
                    (EditText)findViewById(R.id.editMetric);
            EditText desc =
                    (EditText)findViewById(R.id.editDescription);

            // Read from EditText
            String type_value = type.getText();
            String metr_value = metr.getText();
            String desc_value = desc.getText();

            // Set value in SharedPreferences
            preferences.edit().putString("type", type_value).apply();
            preferences.edit().putString("metr", metr_value).apply();
            preferences.edit().putString("desc", desc_value).apply();
        }
    });
}
```

Listing A.13: Saving values from the layout-objects

The data type, metric and description values should now be saved in the shared preferences and can be accessed anywhere in the application.
Appendix B

Contributions to the BITalino Java API

While developing using the Java SDK provided by BITalino, some bugs and shortcomings was discovered. The discovered bugs and shortcomings was fixed as described below and is now a part of the SDK.

B.1 Analog channel always zero

During acquisition from the BITalino board the sample collected from the sixth analog channel always had the value zero. Tests was performed with multiple different sensors. However, the value remained zero. By taking a closer look at the source code of the SDK the source of the bug was located. When decoding the data packet received from the BITalino, the value from the sixth analog channel was only extracted when the size of the data packet was 11 bytes. From the BITalino documentation we know that the maximum size of the data packet is 8 bytes, hence the value of the sixth channel would never be extracted.

1 Java SDK: BitalinoFrameDecoder.
2 if (totalBytes == 11)
3 frame.setAnalog(analogChannels[5], (buffer[j - 7] & 0x3F));

The bug was fixed by comparing against the number of selected channels instead of the calculated number of bytes in each packet.

1 if (analogChannels.length >= 6)
2 frame.setAnalog(analogChannels[5], (buffer[j - 7] & 0x3F));
3 4 https://github.com/BITalinoWorld/java-sdk/pull/1
B.2 Scaling functions

The acquired values are integers between 0 and 1023, by applying scaling functions to the acquired value the value is converted to a more meaningful metric. The SDK provided scaling functions for multiple of the sensor types available for BiTalino, most of the scaling functions relying heavily on the getResolution(int port) method which returns the resolution of the analog channel of a port.

```java
private static final int getResolution(final int port) {
    return port < 4 ? 1023 : 63;
}
```

The getResolution method returns an int, which caused integer division in most of the provided scaling functions. One of the methods affected by the integer division was the scaling function for luminosity. The method would return either 0 or 100, and nothing in between.

```java
public static double scaleLuminosity(final int port, final int raw) {
    return 100 * (raw / getResolution(port));
}
```

By updating the return type of the getResolution method, to float or double, the integer divisions are avoided.

```java
private static final double getResolution(final int port) {
    return (double) port < 4 ? 1023 : 63;
}
```

The accelerometer scaling method contained another integer division bug, this was corrected by casting the numerator to a double.

https://github.com/BITalinoWorld/java-sdk/pull/3

B.3 Enchantments

In addition to the bug fixes above, some enchantments was committed to the SDK. The enchantments are not bugs in the the SDK. However, they might be useful when using the SDK.

B.3.1 Packet sequence check

The BiTalino data packet contains a sequence number between 0-15, for each packet the number is incremented by one. Except in the special scenario when the sequence number is 15, then the next sequence number is 0. Holes in the sequence of sequence numbers indicates
that a packages is missing. If the first packages arriving got sequence number 0 and the second package got sequence number 2, it is clear that the package with sequence number 1 is missing.

To check if the order of incoming sequence numbers are correct the current sequence number was compared with the previous sequence number. If the current sequence number is equal to the previous sequence numbers next the ordering of the sequence numbers are correct. This is checked for each incoming data packet, as shown below where “f” is the data packet.

```
// Initially.
int prevSeq = 15;
int curSeq = 0;

// For each incoming data packet
currentSeq = f.getSequence();
if (currentSeq != (prevSeq+1)%16) {
    Log.d("Sequence check", "missing packet " + prevSeq + ", " + currentSeq);
}
prevSeq = currentSeq;
```

**B.3.2 Added scaling functions**

Since Paolo Pires created the Java SDK for BITalino the BITalino team has added new sensors to their arsenal, namely temperature, EEG and Respiration. Transfers functions for the new sensors are provided in their documentation on the BITalino webpage. The methods implementing the transfer functions was created and are now a part of the Java SDK.

```
/**
 * Temperature conversion.
 * @param port the port where the <tt>raw</tt> value was read from.
 * @param raw the value read.
 * @param celsius <tt>true</tt>:use celsius as metric, <tt>false</tt>: fahrenheit is used.
 * @return a value ranging between -40 and 125 Celsius (-40 and 257 Fahrenheit)
 */
public static double scaleTMP(final int port, final int raw, boolean celsius){
    double result = (((raw/getResolution(port))*VCC) -
0.5) * 100;

    if (!celsius)
        // Convert to fahrenheit
        result = result * ((double) 9 / 5) + 32;

    return new BigDecimal(result).setScale(2,
            RoundingMode.HALF_UP)
            .doubleValue();
}

/**
* Respiration conversion.
* @param port the port where the <tt>raw</tt> value was read from.
* @param raw the value read.
* @return a value ranging between -50% and 50%
*/
public static double scalePZT(final int port, final int raw)
{
    double result = ((raw / getResolution(port)) - 0.5) * 100;
    return new BigDecimal(result).setScale(2,
            RoundingMode.HALF_UP)
            .doubleValue();
}

/**
* Electroencephalography conversion.
* @param port the port where the <tt>raw</tt> value was read from.
* @param raw the value read.
* @return a value ranging between -41.5 and 41.5 microvolt
*/
public static double scaleEEG(final int port, final int raw)
{
    double G_ECG = 40000; // sensor gain
    // result rescaled to microvolt
    double result =
            ((raw / getResolution(port)) - 0.5) * VCC / G_ECG;
    result = result * Math.pow(10, 6);
    return new BigDecimal(result).setScale(2,
            RoundingMode.HALF_UP)
            .doubleValue();
}
https://github.com/BITalinoWorld/java-sdk/pull/4
Appendix C

User manual for prototype

Application user manual This user manual is intended to aid users that want to collect data from a BITalino sensor board, and store the collected data to a file on the mobile device. The user manual covers how to turn on the BITalino sensor board, initial setup of the devices, a step by step explanation of how to collect data, and lastly how to access the collected data. The initial setup is only required to perform the first time a mobile device and a BITalino sensor kit are used together, and can otherwise be skipped. For this we use 2 applications, see Figure C.1. The BITalino sensor wrapper application to collect data from the BITalino and the Collector provider application to store the collect data to the file.

![Figure C.1: Application icons of Collector and BITalino](image)

This user manual is illustrated by following an example. In the example data collecting is performed at 10 Hz with three sensors connected to the BITalino sensor board, two respiration belts and one oximeter. One of the respiration belts are placed around the chest of the monitored person, the second one around the stomach, and the oximeter is placed on the index finger on the patient’s left hand. The first respiration belt is connected to analog input number 1 on the BITalino board, the second respiration belt to analog input number 2, and the oximeter to analog input 3.
C.1 Turning on the BITalino

Before performing the initial setup and/or the data collecting, the BITalino device and the mobile device needs to be fully charged and turned on. To turn on the BITalino sensor board, switch the on/off switch of the BITalino sensor board on, see Figure C.2. This should turn on the LED light on the BITalino sensor board.

![Picture of the BITalino with and without protection case](image)

Figure C.2: Picture of the BITalino with and without protection case

C.2 Initial setup

C.2.1 Pair the devices over bluetooth

In order to collect data with the BITalino it needs to be paired with the mobile device. To do this the user needs to go to the bluetooth settings, turn bluetooth on and search for new devices. The “bitalino” should appear under the tab “Available devices”, see Figure C.3.

1. Press the available “bitalino” device to initialize pairing with the BITalino device. At this point a window asking for the PIN to pair with BITalino should pop up.

2. Enter “1234” and press “OK”. The mobile device should now be paired with the BITalino device, which should now be located under the “Paired devices” tab with the name “bitalino”.

C.2.2 Specify MAC-address of BITalino

The next step is to specify the MAC-address of the BITalino to the BITalino application. This is done to specify which device the mobile device should collect data from. To specify the MAC-address. Launch the BITalino application and scroll to the bottom of the screen.

1. Enter the MAC-address of the into the text field marked with “1” in Figure C.4. The MAC-address of the is located at the back of the BITalino board marked with “MAC”, see Figure C.5.
2. Save the entered MAC-address by pushing the button marked with “2” in Figure C.4. If a valid MAC-address was entered a message displaying “Configuration saved” should appear, if an invalid MAC-address was entered “Invalid MAC-address” will be displayed instead. In the case where the second message is displayed, correct the entered MAC-address and push the button.
C.3 Collect data

At this point the BITalino device should be turned on and paired with the mobile device. The first thing to do is to connect the sensors to the patient and connect them to the BITalino sensor board at any of the inputs, marked with 1 to 6 at the BITalino sensor board.

C.3.1 Configure BITalino

Launch the BITalino application. This application is used to configure which inputs to collect data from at the BITalino sensor board, as well as selecting the number of samples per second.

1. Select “RAW” in the drop down menus, as the one marked with “1” in Figure C.6, with the same numbers as the inputs used to plug sensors into the BITalino sensor board. For the presented example “RAW” is selected for the first three drop down menus, and “<OFF>” for the rest.

2. If desired, enter a description for each of the sensors connected in the description text field, as the one marked with “2” in Figure C.6, for each of the inputs. In Figure C.6 suitable descriptions are given to the three active inputs.
3. Select the desired sampling frequency with the draggable bar marked with “3” in Figure C.6. It is possible to choose 1, 10, 100 or 1000 samples per second. For the presented example 10 is chosen as the sampling frequency.

4. Save the configurations by scrolling to the bottom of the screen and push the button with the text “SAVE CONFIGURATION”. A message displaying “Configuration saved” should appear.

C.3.2 Collecting

Launch the Collector application. This application is used to start the data collecting and specifying where to store the data.

C.3.3 Configure data storage

This step needs to be done once, if you are sure that the storage preferences are as desired this step can be skipped. Push the settings button marked with “1” in Figure C.7. This opens the settings for the Collector application.

1. Make sure that the toggleable button, marked with “1” in Figure C.8, is toggled to “FILE”.

2. Enter the name of the file to save the data to in the text field marked with “2” in Figure C.8.

3. Save the configuration by scrolling to the bottom of the screen and push the button with the text “USE THIS CONFIGURATION”.

Figure C.6: Screenshot of the BITalino application
When pushed the settings are saved and the user interface changes back to the one in Figure C.7.

At this point all the configurations should be finished, and the system is ready to start collecting data.
C.3.4 Start collecting

Connect the sensors to the monitored person and open up the Collector application if it is not already open.

1. Check the checkbox, marked with “2” in Figure C.7, next to the text “BITalino”.

2. Push the button marked with “3” in Figure C.7 to select the checked items. This will change the user interface to the one shown in Figure C.9.

3. Press the “START” button to start the data acquisition. This will change the background color to green to indicate that the data collecting has started. To check if the BITalino has started to collect data successfully check if the LED light on the BITalino sensor board is blinking two times per second.

![Figure C.9: User interface after selecting sensor wrappers](image)

Collect data as long as desired, and stop the collecting by pushing the “STOP” button and turn off the BITalino.

C.4 Access collected data

The collected data is located in a file in the “Download” folder of the mobile device. To access this file connect the mobile device to a computer.

1. Navigate to the Download folder of the device using a file explorer on the computer.
2. Copy the file, and paste it at the desired location on the computer with a meaningful name. From the name of the file it should be possible to deduce the context of the content.

3. (Optional) Delete the file from the phone. To do this perform step 1, and then right click the file and select “Delete”.

If step 3 is not performed between sessions the collected data is appended to the file on the phone. This is fine as long as there is enough free memory on the phone. However, it is recommended to delete the file from the phone after each session to free memory resources.
Appendix D

Python scripts

The prototype provider application developed in this thesis is able to forward the collected data using TCP. This makes it possible to collect data using a mobile device and forward the collected data to a more powerful computer for further computation. Three python scripts are created to show some of the possible use cases enabled by this feature. Moreover, the developed scripts can visualize the data, store it to a database, and perform a simplification of complex event analysis on it.

D.1 Common features

All of the three scripts contain certain common features, such as listening and establishing a TCP connection, reading messages from the connection and recreate JSON objects of these messages. In this section the common features are explained in detail. The scripts takes two command line arguments, the IP-address and port number it should listen for a connection at. If less than two arguments are provided a message telling the user how to use the script is printed to the terminal, as shown in Listing D.1.

```python
# Check if the necessary arguments are provided
if (len(sys.argv) < 3):
  print "Usage: %s <IP-address> <Port>" %sys.argv[0]
  exit(1)
```

Listing D.1: Usage message of scripts

If enough arguments are provided, the command line arguments are used to create a TCP socket by binding to it. Then the socket is used to listen for incoming connection attempts to the specific IP-address and port combination. The `s.accept()` function call is blocking and the subsequent commands are not executed before a connection attempt is accepted. When an attempt is made it is accepted and the resulting socket is stored in the `clientsocket` variable. The user is notified
that a connection is accepted by a printed message to the terminal. The socket (clientsocket) is used to read and write messages to the other communication participant (i.e. the mobile device).

```python
# Read command line arguments
TCP_IP = sys.argv[1]
TCP_PORT = int(sys.argv[2])
# Specifies IP-address and port of the TCP socket
s = socket.socket(socket.AF_INET,
                   socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
# Blocks and listens for a connection
s.listen(1)
print "Server started and listening"
clientsocket, address = s.accept()
# Connected
print "accepted"
```

Listing D.2: Creating a TCP socket

The established connection timeouts if a message is not received within the last 30 seconds, which is done by using a try-except statement. The try-block reads messages from the socket until a timeout is triggered. A timeout is triggered if a message is not received for 30 seconds, and when the timeout is triggered the except block is executed. It closes the sockets to the other communication entity, which makes the port free for others to use.

```python
try:
    while True:
        # Read messages and perform further computations.
    except socket.timeout:
        # No message received in the last 10 seconds, closing connection
        clientsocket.shutdown(socket.SHUT_RDWR)
        clientsocket.close()
        s.shutdown(socket.SHUT_RDWR)
        s.close()
```

Listing D.3: Timeout mechanism of socket

The messages sent by the provider application is newline separated and they are sent one by one. However, the underlying TCP implementation might concat and split messages as it pleases. To cope with this the scripts read chunks of data from the socket and splits the read data on newline characters. When splitting on newline characters the data between the newline characters are placed in a list (data_split).
The last element in this list is either an empty string (i.e. if the last character is a newline character) or a partial message (i.e. if the last character is not a newline character). In either case, the last element is temporary stored and the next read chunk is appended to it. After the read chunk is splitted into the different messages, they are looped through one by one. In this loop the messages are parsed and the performed computations varies between each script.

```python
try:
    # While a message is received within the last 30 seconds
    Data_string = ""
    while True:
        # Read a chunk of data
        data = clientsocket.recv(10000)
        data_len = len(data)
        if data_len > 0:
            data_string += data.decode()  # append to the previous message
            data_split = data_string.split(\'\n\')  # messages are newline separated
            data_string = data_split[-1]  # In case of incomplete message

            for lines in data_split[:-1]:
                # Perform further computations

        except socket.timeout:
            # close connections.
```

Listing D.4: Reading and splitting data into data packets

## D.2 Fall detection

This example script performs a simplification of Complex Event Processing (CEP) on accelerometer values collected by a mobile device. CEP use a sliding window on a data stream, the values inside the window are used to determine if an event has occurred. In this script the window is defined to be the five most recently received data packets and the event is if the phone has fallen. To determine if the phones has fallen a threshold based algorithm is used, like the one proposed in (insert article). The algorithm combines the accelerometer values from each of the three directions to determine the total acceleration of the object. This can be found by calculating the length of the euclidean vector of the accelerometer values $a_1$, $a_2$ and $a_3$ as shown in Figure D.1.

$$||[a_1,a_2,a_3]|| = a_1^2 + a_2^2 + a_3^2$$

(D.1)

Figure D.1: Euclidean length of a vector
The total acceleration is higher when the object is lying still than if the object is in freefall because of gravity. Additionally, the total acceleration is larger if the object has been in freefall and then suddenly stop (i.e. by hitting the ground) than if it is lying still. We can use this to determine if the object has fallen, since a fall is characterized by a freefall followed by a stop. Moreover, if the low threshold for a free fall and the high threshold for a stop is reached within the window a fall has occurred. The implementation parses and extracts the data values for each of the accelerometers from each received data packet. Then the euclidean length is calculated and compared against the thresholds. If a high spike (i.e. the object has hit the ground) is detected and a free fall was detected within the 4 previous data packets the user is alerted by a message printed to standard out.

```python
i = 0
free_fall = False
for lines in data_split[:-1]:
j_data = json.loads(lines)  # Create json-object of string
    if j_data['type'] == "data":  # Calculate the length of the euclidean vector
        euc_len = 0.0
        for elem in j_data['data']:
            euc_len += pow(elem['value'], 2)
        euc_len = sqrt(euc_len)
        if euc_len <= 6.0:  # Phone in free fall
            free_fall = True
            i = 0

        if free_fall == True:  # Free fall within sliding window
            if euc_len >= 13.5:  # Check if phone has hit the ground
                i = 0
                free_fall = False
                print "FALL"

        # End of sliding window
        if i > 4:
            i = 0
            free_fall = False
            i += 1
```

Listing D.5: Threshold-based fall detection algorithm

The implementation assumes that each data packet contains exactly three accelerometer values. However, a better solution would be to add some checks to only accept the data packets that contain exactly three accelerometer values and discard the rest. This example scripts shows that it is possible to the perform real-time analysis of the data on a
computer. Unfortunately, it could lead to a large amount of network traffic. However, if the resources on the mobile device is limited, this makes it possible to perform real-time analysis by moving the computational work away from the mobile device.

D.3 Database storage

This example scripts reads data packets from the socket and stores all the information they contain to a SQLite database. It is intended as an example of how to use the combination wrapper ID and channel ID to link data packets with its metadata. Therefore, the metadata is temporary stored in matrixes, and whenever a data packet is received the data values of each data channel is combined with its metadata and saved to the database. To ease the creation and insertion into the database and matrices the helper functions is Listing D.6 are used.

```python
1 def open_data(con):
2    cur = con.cursor()
3    try:
4        cur.execute("CREATE TABLE SensorData(WrapperID INT, WrapperName TEXT, Timestamp TEXT, ChannelId INT, Value FLOAT, Datatype TEXT, Metric TEXT, Description TEXT)"
5                      
6        except:
7            print "Allready created"
8    return cur
9
10 def insert_data(cur, w_id, w_name, time, c_id, value, datatype, metric, des):
11    cur.execute("INSERT INTO SensorData VALUES(%d,'%s','%s',%d,%f,'%s','%s','%s')" % (w_id,
12                                                                                     w_name, time, c_id, value, datatype, metric, des))
13
14 def create_string_matrix(x, y):
15    matrix =[['']*y for x in range(x)]
16    return matrix
```

Listing D.6: Helper functions for database script

First, the script calls the open_data() function to create a database, then create_string_matrix(x, y) is used to create matrices to store the metric, data type and description of each combination of wrapper ID and channel ID. Then the script parses each received data packet, and calls different functions depending on the type of the data packet.
# Open database
con = lite.connect('sensordata.db')
cur = open_data(con)

# Create matrices
wrapper_name = [['' for k in range(10)] for i in range(10)]
meta_type = create_string_matrix(10,10)
meta_metric = create_string_matrix(10,10)
meta_descr = create_string_matrix(10,10)

try:
    while True:
        # read chunks of data
        ...
        for line in data_split[:-1]:
            json_data = json.loads(line)
            if json_data['type'] == "meta":
                save_metadata(json_data)
            if json_data['type'] == "data":
                save_data(json_data)
except socket.timeout:
    # Close connections

Listing D.7: Main loop of the database script

If a metadata data packet is received the save_metadata() function is called, it parses the data packet and stores the metadata in the corresponding matrix, as shown in Listing D.8. The wrapper ID and channel ID is used to determine the position of each value in the matrixes; the wrapper ID determines the x-position and the channel ID determines the y-position.

def save_metadata(data):
    wrapper_id = int(data['id'])
    wrapper_name[wrapper_id] = data['name']
    for channel in data['channels']:
        channel_id = int(channel['id'])
        channel_type = channel['type']
        channel_metric = channel['metric']
        channel_descr = channel['description']
        meta_type[wrapper_id][channel_id] = channel_type
        meta_metric[wrapper_id][channel_id] = channel_metric
        meta_descr[wrapper_id][channel_id] = channel

Listing D.8: Function for saving the metadata to matrices

If a data packet is received the save_data() function is called. It parses the data packet and combines the data values and the metadata of each channel using the combination of wrapper ID and channel ID.
def save_data(data):
    w_id = int(data[‘id’])
    time = data[‘time’]
    for elem in data[‘data’]:
        c_id = int(elem[‘id’])
        insert_data(cur, w_id, wrapper_name[w_id], time,
                    c_id, elem[‘value’],
                    meta_type[w_id][c_id], meta_metric[w_id][c_id],
                    meta_descr[w_id][c_id])

Listing D.9: Function for saving data to the database

When all of the data is stored in the database one could for instance apply data mining algorithms on the data (i.e. to detect OSA) or one could perform simpler queries. In an experiment similar to the overnight experiment described in Section 6.2, data values were collected during a whole night using a BITalino sensor board. The collected values were forwarded to a computer, which saved the data to a database, as described in this section. The BITalino sensor board was given the wrapper ID of 0, and the third analog channel of the BITalino was given the channel ID 2. A luminous sensor was connected to this analog input. To obtain the values collected by this sensor the query in Listing D.10 can be used.

```
SELECT Value FROM SensorData WHERE ChannelId=2 AND WrapperId=0;
```

Listing D.10: Query for obtaining luminosity values

We can then dump the results of the query to a file and plot the obtained values. The resulting graph is shown in Figure D.2, where we can see that the light located over the sensor was turned on at the beginning of the acquisition. The light was then turned off, and at the end of the acquisition period we can see that the sun is starting to shine in through the window resulting in increasing values.

## D.4 Visualization

The third example script enables real-time visualization of the collected data by using a third-party plotting tool. The tool used is called Kst (version 2) [52] and is licensed under GPU [53], as such it is freely available for anyone to use. Kst is a fast and robust plotting tool, which
includes many useful features to manipulate the plots (such as to show all values or to only show the n newest values). Data can be provided to Kst by using a file containing the data values. This file is read line by line, and each line contains whitespace separated data values, as shown in Listing D.11.

Listing D.11: Structure of the shared file

The Python scripts writes the received values to this file, which Kst reads from, creating a kind of shared memory between them. As soon as a new line of data is written to the file it is plotted by Kst. The implementation of the script parses each received data packet and extracts the data values of each data channel. These values are then combined to a string, which is written to the shared file.
```python
outfile = open('shared_file.txt', 'w')
data_string = ""
try:
    while True:
        # Read chunk of data and split into messages
        for line in data_split[:-1]:
            json_data = json.loads(line)
            if json_data['type'] == "data":  
                # Create string and write it to file
                string = ""
                for elem in json_data['data']:
                    string += "%f " % elem['value']
                outfile.write(string + "\n")
                outfile.flush()
        except socket.timeout:
            outfile.close()
    # Close connections.
```

Listing D.12: Main loop of the plot script

To start the visualization the script has to be started first, otherwise Kst will not find the file containing the data values. Once the script is started (and the file created), Kst can be started. This is done by using the command shown in Listing D.13. The number of command line arguments depends on how many data values each line in the file contain. If each line contains six data values (such as Listing D.11).

```
('> kst2 shared_file.txt -y 1 -y 2 -y 3 -y 4 -y 5 -y 6
```

Listing D.13: Terminal command to start Kst

This will plot the data values contained in the file, and will continuously plot new data values as soon as they are written to the file. Starting Kst with the command shown in Listing D.13 will result in a plot as the one shown in Figure D.3, which is customized with the GUI to shown the 200 most recent data values of each subplot.
Figure D.3: Screen shot of Kst
Bibliography


