Nidra: An Extensible Android Application for Recording, Sharing and Analyzing Breathing Data

An Engineering Approach

Jagat Deep Singh

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Autumn 2019

Nidra: An Extensible Android Application for Recording, Sharing and Analyzing Breathing Data

An Engineering Approach

Jagat Deep Singh

Nidra: An Extensible Android Application for Recording, Sharing and Analyzing Breathing Data

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

A vast majority of medical examinations requires the patient to be present at the hospital or laboratory. Statistics Norway [39] presents that between 2017-2018 the cost of diagnosis, treatment, and rehabilitation in Norway increased with 7.3 percent for municipal health service. Likewise, the man-years for physicians in the municipal health service increased with 2.4 percent. As such, the growth of medical attendance results in more work and stress induced on the physicians and a higher demand for medical attention from the patients.

Mobile applications that focus on improving healthcare are known as mHealth applications [31]. An excellent example of a mHealth application is the CESAR project, which aims to use low-cost sensor kits to monitor physiological signals during sleep in order to provide early detection of obstructive sleep apnea (OSA) from home [4]. The project facilitates tools that provide a common interface for sensor sources with different Link Layer technology (e.g., BlueTooth, USB, and ANT+) and sensor-specific protocols designed by the manufacturer.

In this thesis, we extend the project by designing and implementing an Android application for users to record, share, and analyze breathing data collected over an extended period. Also, extending one of the tools to support the Flow sensor, which is a respiratory belt for measuring breathing. The motivation is to collect breathing data that can aid in the analysis and early detection of sleep apnea; albeit, the application can be used in other fields of study (e.g., physical activities). Additionally, we facilitate an extensible application, that allows for future developers to create modules that extend the functionality in the application or enrich the data from the user's records. The name of the application is Nidra—named after the Hindu goddess of sleep.

Experiments and observations have shown that the application is capable of collecting data over an extended period using the Flow sensor. More specifically, 9-hours of recording use approximately 1395 mAh of the battery capacity, which is well sufficient based on the average battery capacity of mobile devices. Also, the application is able to reconnect with the sensor upon disconnects—with the results from the experiment, worst-case shows no more than 40% data loss. Finally, other developers are successfully able to create modules that integrate with Nidra.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Dr. Thomas Peter Plagemann, for his guidance throughout the work in this thesis. The discussions between us have given me invaluable insights that have been helpful during the development and writing of this thesis. For that, I am genuinely grateful for the effort and dedication he put into helping me.

Next, I'd like to thank my friends that I've gained during the study, as well as my childhood friends. Their encouragement and motivation have kept me going, through thick and thin.

Above all, I'm truly grateful for my parents' unconditional love and care. Without the support and affection from my Mom, I would not be where I am today. To her, this degree means much more than it does to me.

A man is but the product of his thoughts; what he thinks, he becomes.

Mahatma Gandhi

Contents

List of Tables

List of Figures

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Background and Motivation

The medical focus in this thesis is in the field of sleep-related breathing disorders, which is characterized by abnormal respiration during sleep. Obstructive sleep apnea (OSA) is a disorder, which in layman's terms is when the natural breathing cycle is partially or completely obstructed in repetitive episodes during sleep. As a consequence, OSA decreases the quality of life, and untreated OSA can lead to severe illnesses like cardiovascular diseases, including diabetes, strokes, and atrial fibrillation [48]. The diagnosing of OSA is performed with polysomnography in sleep laboratories. However, this method is both an expensive and time-consuming procedure that takes a toll on the patient and the laboratories. The patient is strapped in a contraption of sensors and ordered to sleep overnight, while the laboratories have limited capacity and resources to perform sufficient tests with the patients.

In recent years, mobile phones have become significantly advanced and powerful devices. What would require an entire room of processing power, has been compressed into a handheld and portable device. As of now, mobile phones come with powerful processors, a sufficient amount of RAM, and an adequate amount of battery capacity. Mobile phones operate with an operating system at its core; the operating system facilitates a platform that enables developers to create and develop applications that can be used by end-users. Moreover, mobile devices come with sensors (e.g., microphones and accelerometers), with the capability of connecting to external sensors through wired or wireless communication channels (e.g., BlueTooth). As such, third-party vendors can create sensors that aid in the detection of events or changes in an environment, and send the information to mobile devices. With respect to monitoring sleep-related breathing disorders, there are various sensor vendors which translate physiological signals (from the human body) into digital signals which can be processed by the mobile device. An example of such a sensor vendor is SweetZpot Inc. [50], which provides an affordable respiratory effort belt that captures

the respiratory effort using strain-gauges, called Flow. The Flow sensor is initially designed to be used during physical activities (e.g., cycling, lifting, and rowing); however, it can be used for collecting breathing data over an extended period, as it is battery-powered and has BlueTooth support.

Creative use of mobile devices and sensor technologies (mHealth) has the potential to improve health research and reduce the cost of healthcare. Mobile technologies[1] can overcome the hurdle of a patients presence, encourage behaviors to prevent or reduce health problems, and provide personalized/on-demand interventions [31]. A potential mHealth application is the CESAR project, which aims to use low-cost sensor kits to monitor physiological signals (e.g., related to heart rate, brain activity, the oxygen level in the blood) during sleep in order to provide early detection and analysis of obstructive sleep apnea (OSA). The CESAR project [4] focuses on the development of new software solutions using state-of-the-art consumer electronic devices with appropriate sensors sources to enable anyone to monitor physiological parameters that are relevant for OSA monitoring at home. This unfolds the potential to provide early detection of OSA from home with the aid of various sensors sources.

As of now, the CESAR project facilities for tools that manage the connectivity with sensor source, and forwarding of data packets from these sensor sources to subscribing applications. More specifically, the data streams dispatching tool [12] manage applications (subscribers) that listen for data packets from sensor sources (publishers). Moreover, the data acquisition tool [25] facilitates for a sensor wrapper that provides a common interface for sensor sources with different Link Layer technology (e.g., USB, BlueTooth, and ANT+) and sensor-specific protocols (e.g., provider defined SDKs), and acts as a publisher that is connected with the data streams dispatching tool.

The motivation for this thesis is to expand on the CESAR project by creating an application, called Nidra, for the patients to record, share, and analyze breathing data. Nidra should be able to collect breathing data over an extended period, share the recording data across applications using a media (e.g., e-mail), and to analyze the data. Additionally, Nidra should support the integration of other standalone applications (i.e., modules) which leverages the data from Nidra in order to enrich the data or extend the functionality of Nidra. For example, a module can be using the data from Nidra to feed a machine-learning algorithm that predicts sleep apnea. In the end, the data from the patients should aid the researchers/doctors in the diagnosis of obstructive sleep apnea. However, we can also apply Nidra in other fields of study (e.g., physical activities).

[1] mobile device and sensors that are intended to be worn, carried or accessed by individuals

1.2 Problem Statement

The market has new and affordable sensors that can aid with the data acquisition, which we seamlessly can integrate with the extensible data streams dispatching tool. The Flow sensor is an interesting sensor source due to its versatility and adaptability of collecting breathing data and connecting to devices with the BlueTooth protocol.

As for the state of this thesis: (i) applications which supports the Flow sensor has not been designed for end-users, in essence, they provide no user-friendly interface that allows for sharing of the data. In order to extract the data from these applications, the mobile device has to be connected with a PC over USB for data transfer; (ii) there is no sensor wrappers that support the Flow sensor with the data streams dispatching tool in the CESAR project; and (iii) the data streams dispatching tool is not ready to be used by the end-users, because the project facilitates no user-friendly interface for users to record the data from the supported sensor sources.

As such, we look into designing and implementing an Android application (Nidra) that record, share, and analyze breathing data over an extended period (e.g., during sleep) by using the Flow sensor. Additionally, we want to facilitate an extensible application such that future developers can extend functionality or enrich the data in Nidra. In the end, we can hopefully strengthen the analysis of abnormal sleeping patterns to decreasing the risk of the symptoms they may come as a consequence. Also, as a bi-product, the application can be used in other fields of studies (e.g., physical activities). As the scope of this thesis, we focus on the completion of three main goals:

Goal 1 Integrate the support for Flow sensor by creating a sensor wrapper that connects with the extensible data streams dispatching tool.

Goal 2 Research and develop a user-friendly application which facilitates collection of breathing data with the Flow sensor, sharing of the data, and analysis of the data with the use of the extensible data streams dispatching tool.

Goal 3 Create an extensible solution such that the developers can create standalone applications that integrate with Nidra.

As part of the goals of this thesis, we also define three system requirements to keep in mind while designing and implementing the application. The three system requirements are the following:

Requirement 1 The application must provide an interface for the patient to (i) record physiological signals (e.g., breathing data); (ii) present the results; and (iii) share the results.

Requirement 2 The application must ensure that upon sensor disconnections, the connectivity is reinstated to minimize the data loss and its effects on the data analysis.

Requirement 3 The application must provide an interface for the developers to create modules that integrate with the application.

1.3 Limitations

Based on the goals and requirements stated in the previous section, the scope of this thesis is to design and implement an application capable of recording breathing data obtained by the Flow sensor over an extended period.

We limit the scope to integrating the support for the Flow sensor in Nidra, and excluded to test for already integrated sensor sources (e.g., BITalino). Further, with the Flow sensor provided under development, we restricted the design to collect respiration (breathing) data (as opposed to hearth-rate or other physiological data).

The application is designed to collect breathing data; we do not perform any analysis to predict or detect sleeping disorders based on the data. However, we facilitate for future developers to utilize the data provided by Nidra to perform such tasks.

Finally, the implementation is Android-specific as the previous work performed on the project is designed solely for Android applications.

1.4 Research Method

The work in this thesis is classified as *computing research* with a principle approach of an *engineering method* as described in [26]. The engineering method (evolutionary paradigm) is to: (i) observe existing methods, (ii) propose a better solution, (iii) build or develop artifacts[2], and (iv) measure and analyze until no further improvements are possible. The report identifies patterns amongst various principle approaches and categorizes the patterns into phases: *(i) informational phase*, *(ii) propositional phase*, *(iii) analytical phase*, and *(iv) evaluation phase*. Below, we give a brief description of each phase and discuss how our work fits into each of them.

1.4.1 Informational Phase

The informational phase according to the report is to *"gather or aggregate information via reflection, literature survey, people/organization survey, or poll"*

In this thesis, we survey previous related work in the field of detecting, analyzing, and diagnosing sleep related-breathing disorders on a mobile device. Based on this, we derive that the application created in this thesis

[2]human-manufactured objects produced during development

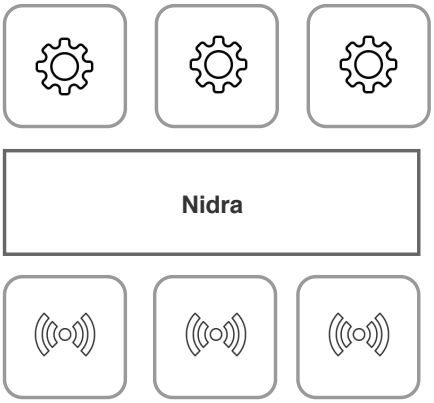

Figure 1.1: Nidra with the possibility of adding multiple modules that extends the functionality or provide data enrichiment, and integrating support for multiple sensor sources (with the use of the data streams dispatching tool).

has the same motivation; however, the related work operates with different of kind measure and instrument for solutions (e.g., using microphone and accelerometers to provide early-detection of sleep apnea). As such, we focused on creating an extensible application that allows future developers to create modules on top of our solution (as illustrated in Figure 1.1). By allowing this, future developers can expedite the innovation in the research and study of sleep-related breathing disorders, as well as allowing the patients to operate with one application.

1.4.2 Propositional Phase

The propositional phase according to the report is to *"propose and/or formulate a hypothesis, method or algorithm, model, theory, or solution"*

The solution proposition in this thesis is to create an application used to record, share, and analyze breathing data collected over an extended period. We want to extend the CESAR project by providing an user-interface to the patient to perform these tasks while using the tools that the project facilitates. Mainly, we want to use the data streams dispatching tool in order to manage current and future sensor sources. In which, we proceed to add support for the Flow sensor. In the end, we wish to facilitate an application that is used by the patients to record their breathing data during sleep and to share the data between researchers/doctors. As such, we aid in collecting breathing data used for early detection of sleep-related breathing disorders (e.g., obstructive sleep apnea) from home.

1.4.3 Analytical Phase

The analytical phase according to the report is to *"analyze and explore a proposition, leading to a demonstration and/or formulation of a principle or theory"*

With the proposition phase of this thesis, we analyze the tasks of the application. We separate the tasks into concerns (i.e., recording, sharing, analyzing, modules, storage, and presentation) where we propose a structure that contains components, where each component has various functionalities and design choices. With each concern combined, we fulfill the goals of this thesis. As a demonstration, we realize the design choices by implementing them as an Android application, called Nidra.

1.4.4 Evaluation Phase

The evaluation phase according to the report is to *"evaluate a proposition or analytic finding by means of experimentation (controlled) or observation (uncontrolled, such as a case study or protocol analysis), perhaps leading to a substantiated model, principle, or theory"*

Based on the requirements and goal of this thesis, we evaluate the application by conducting experiments. Some of the experiments include participants that perform various tasks, such that we can observe the outcome of the tasks on participants without prior knowledge or experience of the application. In the end, we evaluate and conclude whether the goals and requirements of this thesis are satisfied.

1.5 Thesis Outline

The thesis is divided into three parts, which the following list presents a general overview of:

- Part 1: **Introduction & Background**

  Chapter 2: Background presents the background material necessary for understanding the fundamentals of this thesis. It starts by introducing the CESAR project and the tools provided for data acquisition, as well as a description of the Flow sensor. Finally, an overview of the Android operating with the information required to understand the structure of the application (Nidra).

  Chapter 3: Related Work presents the related work focusing on creating a mobile application to collect physiological data in order to diagnose sleep apnea, and with a brief discussion on how we contribute with novelty and improvements from the related work.

- Part 2: **Design & Implementation**

  Chapter 4: Analysis and High-Level Design presents the functional requirements of the application, the tasks derived based on the requirements and goals of the thesis, and the tasks separated into concerns, where we propose a structure of implementation which contains component with various functionalities and design choices. In the end, we discuss the data structure, namely the data entities (i.e., record, sample, module, and user), the data format (JSON versus XML), and the structure of the data packets sent from the sensor sources as well as the data packets sent on sharing.

  Chapter 5: Implementation presents the application components of the project (i.e., Nidra, data streams dispatching module, and the Flow sensor wrapper) with the interface of IPC connectivity. Moreover, it presents the steps and flows taken in order to implement the concerns (i.e., recording, sharing, analyzing, modules, storage, and presentation) by separating the actions and showing how the components in the application interact.

- Part 3: **Evaluation & Conclusion**

  Chapter 6: Evaluation presents four experiments conducted in order to evaluate the system requirements of the application. Each experiment has a short description followed up by results, a discussion on improvements and findings, and a conclusion of the experiment. Finally, a conclusion on the system requirements of the thesis.

  Chapter 7: Conclusion presents the summary of the thesis, followed up with contributions that answer the goals defined in the problem statement. Finally, a list of improvements and furture work that can be made to the application.

Chapter 2

Background

2.1 The CESAR Project

The goal of the CESAR project is to reduce the threshold to perform a clinical diagnosis of obstructive sleep apnea (OSA) and to reduce the time to diagnose the disorder [13]. Obstructive sleep apnea is a common sleeping disorder which affects the natural breathing cycle by reducing respiration or all airflow. As a consequence, OSA can lead to serious health implications, and in some cases, death through suffocation. Moreover, the project aims to use low-cost sensor kit to prototype applications using physiological signals related to heart rate, brain activity, the oxygen level in the blood to monitor sleep and breathing-related illnesses [4].

As of now, the development in the project facilities for data acquisition from various sensor sources and forwarding of the data packets to subscribed applications. Section 2.1.1 and Section 2.1.2 is a summary and results of the development on the project. The previous work has incorporated support for a few sensor sources (e.g., BITalino) in order to collect physiological signals. With the support for more sensor sources allows for more precise detection and analysis of the disorder. In Section 2.1.3, a new affordable sensor source is introduced.

As an overview, Figure 2.1 is an illustration of the structure for the CESAR project. The system is divided into three parts: data acquisition part, data streams dispatching part, application part. The first part enables developers to create sensor wrappers that connect with sensor sources through different Link Layer technology (e.g., Ethernet, USB, Bluetooth, WiFi, ANT+, and ZigBee). The second part manages the discovery and collection of these sensor wrappers and package distribution to subscribing applications. The third part uses the interface that the second part provides in order to collect data from the desired sensor sources.

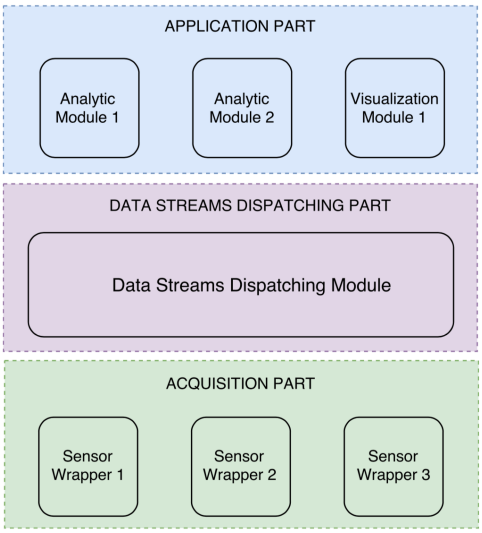

Figure 2.1: Structure of the CESAR project, separating functionality into three independent parts [12].

2.1.1 Extensible Data Acquisition Tool

In the thesis "Extensible data acquisition tool for Android" by Gjøby [25] a proposition of a *data acquisition system* for Android is presented. The thesis proposes a system that hides the low-level sensor-specific details into two components, *providers* and *sensor wrappers*. The provider is responsible for the functionality that is common for all data sources (e.g., starting and stopping the data acquisition), while the sensor wrapper is responsible for the data source specific functionality (e.g., communicating with the data source).

The thesis solves the difficulties around creating an extensible data acquisition tool, connecting new and existing sensors, and finding a common interface. The problem statement of the thesis addresses the following concerns regarding sensors:

- *Common abstraction/interface for the interchanged data*
  Sensor platform manufacturers have their low-level protocol to support the functionality of their product. Typically, the manufacturers provide software development kits (SDKs) to hide the low-level protocols so third-party developers can be easier; however, both the low-level protocol and the SDKs are not standardized. Thus, for each sensor, there might be different commands and methods.

- *Various Link Layer technologies*
  Each sensor might use different Link Layer technology (i.e., Ethernet, USB, BlueTooth, WiFi, ANT+, and ZigBee), which means establishing a connection between a device and a sensor might differ. For instance, BlueTooth devices need to be paired, while devices on the WiFi can address each other without any pairing.

- *Reusability of sensor code*
  Applications that implement support for the low-level protocol of a sensor type can not be shared between different applications. Thus, introducing duplicate work and code if multiple application wish to use the same sensor type. A framework that isolates the sensor that applications can use, might make it easier for application to utilize the collected data. In addition, isolating the sensors into modules improves the robustness and quality of the implementation.

In the thesis, the goal is to develop an extensible system, which enables applications to collect data from various external and built-in sensors through one common interface. The solution around an extensible system is to have the core of the application unchangeable when adding support for a new data source, regardless of the Link Layer technology and communication protocol used by the data source. Making all the data sources behave as the same, is a naive solution to the problem. However, separating the software into two different components, a *provider* component and a *sensor wrapper* component, enables the reuse of functionality that is common amongst the data sources.

The sensor wrapper application is tailored to suit the Link Layer technology and data exchange protocol of one particular data source. Additionally, responsible for connectivity and communication with the data source. The provider application is responsible for managing the sensor wrappers—starting and stopping the data acquisition—and processing the data received from the sensor wrapper application. Thus, everything that is independents of the data source should be a part of the provider application. With this type of solution, the possibility to reuse the sensor wrapper application for different provider applications is made possible. However, there are some overheads with this solution. Mostly, the inter-process communication that might be costly and increase the complexity of the code. Nonetheless, the flexibility and extensibility gained by separating the functionalists out weights the cost.

When a connection is established with the provider application, a package of metrics and data type (all the data does not change during the acquisition as metadata) is sent describing the context of the data collected. The metadata is necessary because different sensors might sample data in different environments, and some applications are depending on recognizing the environment of the data acquisition. Therefore, it is critical to know what data values are measured. Consequently, exposing sensors and data channels through one common interface requires a field of metadata which can be used to: (1) *distinguish* sensor wrapper and data channel the data originated from; (2) determine the *capabilities* of the sensors (i.e., EEG, ECG, LUX); (3) determine the *unit* the data is represented in (i.e., for temperature, Celsius or Fahrenheit); (4) describing the data channel (i.e., placement of the sensor); and (5) a timestamp of when the data was sampled.

To summarize, the task of a *sensor wrapper* is to establish a connection

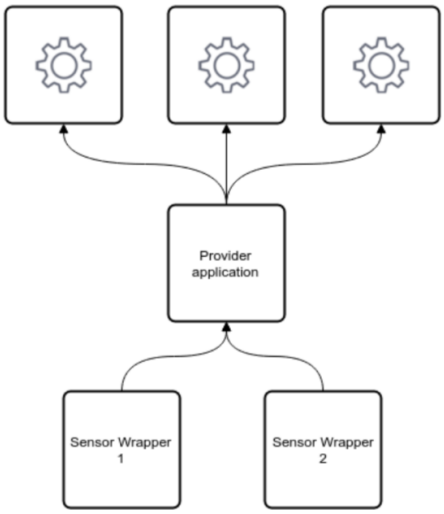

Figure 2.2: Sharing the collected data between multiple applications [25].

to and collect data from precisely one specified data source, and to send the collected data to the *provider application* that is listening for it. A data source (e.g., BiTalion) can have support for multiple sensor attachments (defined as data channel in the thesis), although, only on sensor wrapper is necessary for each data source and their data channels. Each sensor wrapper is tailored to adapt to the data source's Link Layer technology and the communication protocol of a respective data source. Upon activation by a provider application, the data is collected by the sensor wrapper, and pushed to the provider application in a JSON-format. An illustration of the structure is found in Figure 2.2.

2.1.2 Extensible Data Streams Dispatching Tool

The extensible data acquisition tool developed by Gjøby [25] leaves some space for improvements. Such improvements are discussed in the thesis "Extensible data streams dispatching tool for Android" by Bugajski [12]. The thesis analyses the potential improvements of the data acquisition tools, which can be extracted into:

- *Lack of reusability*
  Only the components that have started the collection can receive the data, and no other components can access the collected data.
- *Lack of sharing*
  Components that perform specific analysis on the collected data in real-time have no way of share the results of the analysis such that other components can use them.
- *Lack of tuning*
  It is not allowed to change the frequency of collection after the start.

Thus, the user has to stop the collection and manually change the frequency of the sensor and then restart the collection.

- *Lack of customization*
  The set of channels cannot be changed during a collection, and the collector receives data from all channels even if it needs only one of them. Thus, the data packet size and resource usage become larger than necessary.

In the thesis, the modularity of the architecture is improved by finding a common model for all available data channels, developing a mechanism for cloning a data packet to allow reusing of data across modules, and allowing the modules to have support for choosing channels they want to receive data from and publish their data to. In the model, these components are distinguished as:

1. *sensor-capability model*: is a representation of all distinct data types and contains all information about the channel. A sensor board usually reads and sends different type of data to a mobile device. Thus, this module is used to control every available data type, such that they can be accessed from the application part at any time.

2. *demultiplexer* (DMUX): is a data cloner, that receives data packets from one input (e.g., from one channel), and duplicates the data several times based on the number of subscribers.

3. *publish-subscribe* mechanism: is an interface responsible for managing requests from subscribers or publishers, also, to be able to terminate these statuses. Additionally, every module from the application will be able to see all capabilities represented by this component, enabling the option to choose a frequency the data collection.

The combination of how these modules cooperate and communicate with each other affects the modularity and performance of the architecture. In the thesis, there are various proposed solutions. The naive solution was to fit all of the elements into each respective sensor wrapper, thus, prioritizing the performance and low resource usage, but making it impossible to distinguish data type from two sensor wrapper with the same data type. This is optimal for the cases where collection only occurs on one sensor board. An improved solution includes to place the demux between the application part and the wrapper layer and to insert the remaining elements in the respective application part. This solution resolves the overload of sensor wrappers performing other tasks besides collecting data; thus, the wrappers are untouched, and they send the data to the demux. However, there are several issues with this solution, e.g., due to various obstacles such as (1) every application module has to configure its sensor-capability model; (2) filter requested data packets from all the channels; (3) and deal with the collection speed on its own.

Addressing these issues leads to the final architecture, which is presented in Figure 2.3, and meets the demands identified in the requirements.

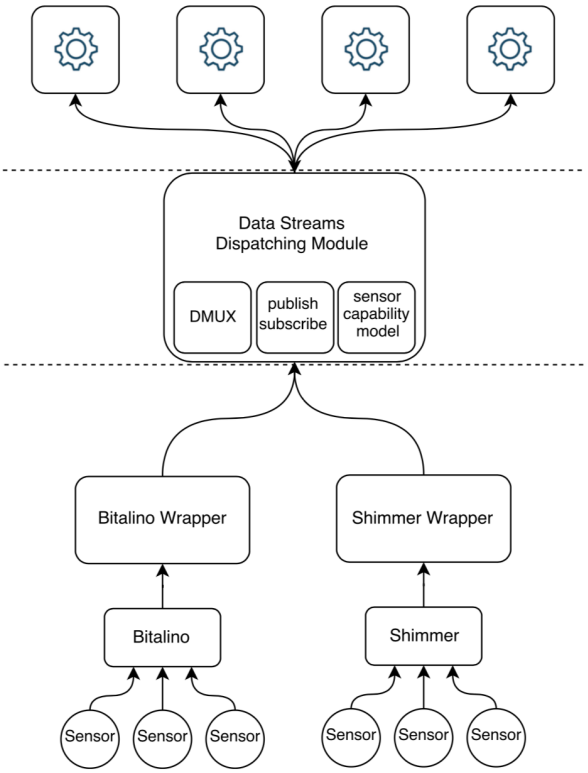

Figure 2.3: Sharing the collected data between multiple applications [12].

In this solution, all elements are placed between the application part and the wrapper layer, forming the data streams dispatching module. The sensor wrapper connects directly to the data streams dispatching module; the module discovers all installed wrappers and populates the sensor-capability model with the data types from all installed sensors. By this, all applications can access a shared sensor-capability mode. A publish-subscribe mechanism enables application modules to subscribe to any capabilities with a preferred sampling rate. Correspondingly, an application module can publish data to other applications through the same interface. The demultiplexing element creates for each subscriber a copy of the data packet.

To summarize, these three elements together establish *the data streams dispatching module*. The final architecture has a couple of advantages, such as it is exceedingly extensible due to its maintainability. For instance, all communication with other layers occurs through one interface; this way, new instances can be added at any time, without the need for modifying large parts of the system. The system is also efficient due to packets are immediately sent to the application on request (without any buffers), and packets are only sent to the application requesting them, resulting in less resource and power usage, and more battery life. This tool is an improvement to Gjøby's solutions due to these facts.

2.1.3 Flow Sensor

Flow is an activity sensor created by SweetZpot Inc., initially designed for measuring breathing and heart rate during activities. The sensor measures breathing per minute, in correlation to the heart rate, for optimal activity measurement. As stated on their page: *"usually, at rest, your breathing varies between 6 and 8 liters per minute. During sleep, breathing can be as low as 3 liters per minute and can reach 160 liters per minute and above during high-intensity athletic activity"* [50]. Thus, this sensor could be suiting for measuring sleeping problems in the project.

The sensor is a strap-on placed beneath the chest. It weighs 27 grams and dimensions of 77x43x17mm. Additionally, it is equipped with a 3V Lithium battery, with an estimated battery life of one year (with 7 hours a weekly usage). The sensor can be connected with BlueTooth technology, making it possible to connect with mobile devices.

2.2 Android OS

Android is an open-source operating system (OS) developed by Google Inc., based on the Linux kernel and primarily designed for mobile touch-screen devices (e.g., smartphones, tablets, and watches). This chapter encompasses the structure of Android, such as the OS architecture and the BlueTooth LE protocol.

2.2.1 Android Architecture

The Android platform is an open-source and Linux-based software stack, containing six major components [42] (as illustrated in 2.4):

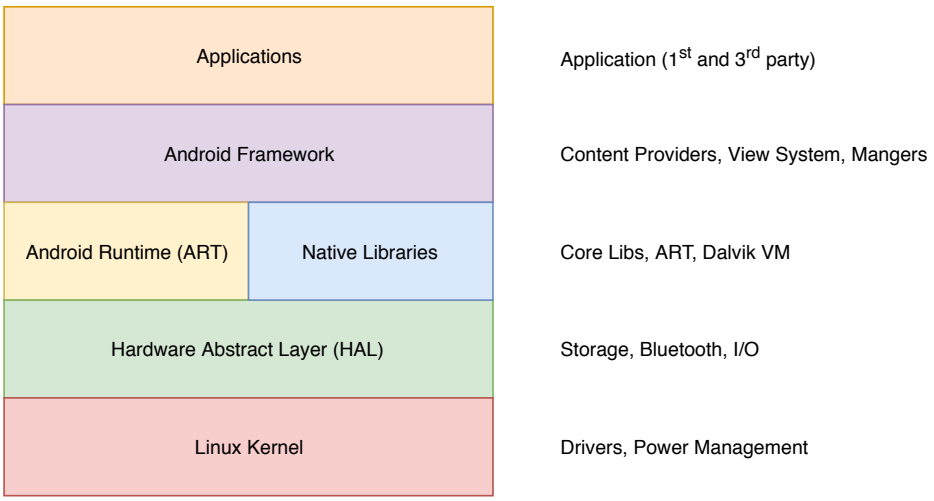

Figure 2.4: Android architecture stack, containing six major components.

Applications: Android provides a core set of applications (e.g., SMS, Mail, and browser) pre-installed on the device. There is support for installing third-party applications, which allows users to install applications developed by external vendors. A user is not bound to use the pre-installed applications for a service (e.g., SMS), and can choose the desired applications for service. Also, third-party applications can invoke the functionality of the core applications (e.g., SMS), instead of developing the functionality from scratch.

Android Framework: Is the building blocks to create Android applications by utilizing the core, all exposed through an API. The API enables reuse of core, modular system components, and services; briefly characterized as *View System*: to build the user interface pre-defined components (e.g., lists, grids, and buttons); *Resource Manager*: provides access to resources (e.g., strings, graphics and layout files); *Notification Managers*: allows applications to show custom notifications in the status bar; *Activity Manager*: manages lifecycle of the application; and *Content Providers*: enables applications to access data from other applications.

Android Runtime: Applications run in a seperate process and has its separate instance of the Android Runtime (ART). ART is designed to run on multiple virtual machines by executing DEX (Dalvik Executable format) files, which is a bytecode specifically for Android to optimize memory footprint. Some of the features that ART provides are ahead-of-time (AOT) and just-in-time (JIT) compilation, garbage collection (GC), and debugging support (e.g., sampling profiler, diagnostic exceptions, and crash reporting).

Native Libraries: Most of the core Android components and services native code, that requires native libraries, is written in C or C++. The Android platform exposes Java APIs to some of the functionality of the native libraries (with Android NDK).

Hardware Abstract Layer: Provides an interface to expose hardware capabilities to the Java API framework. Hardware Abstract Layer (HAL) consists of multiple library modules that implement an interface for specific hardware components (e.g., camera, or BlueTooth module).

Linux Kernel: is the foundation of the Android platform. The ART relies on the functionality from the Linux kernel, such as threading and low-level memory management. The Linux kernel provides drivers to services (e.g., BlueTooth, WiFi, and IPC), and incorporates a component for power management.

2.2.2 Application Components

Application components consist of four core components that are the building blocks of an Android application [7]. This section intro-

duces these components; `Activites`, `Services`, `BroadcastReceivers`, and `Content Providers`. The activity is responsible for interactions with the user, services is a component that performs (long-running) tasks in the background, broadcast receivers handle broadcast messages from application components, and content providers manage shared set of application data. The Android system must be aware of the existence of the components, which can be accomplished by defining a manifest file (`AndroidManifest.xml`) that describes the component and the interactions between them, as well as describe the permission of the application.

2.2.2.1 Activity

An application can consist of multiple activities, and activity represents a single screen with a user interface [2]. Applications with multiple activities have to mark one of the activities as the main activity, which will be presented to the user on launch. The user interface of an activity is constructed in layout files which define the interaction logic of the user interface, and the layout file is inflated into the activity on launch.

Activites are placed on a stack, and the activity on top of the stack becomes the running activity. Previous activities remain in the stack (unless disregarded), and are brought back if desired. An activity can exist in three states:

- **Resumed (Running)**: The activity is in the foreground of the screen and has user focus.
- **Paused**: Another activity is running, but the paused activity is still visible. For instance, the other activity does not cover the whole screen. A paused activity maintains its state but can be killed by the system if the memory situation is critical.
- **Stopped**: Another activity obscures the activity. A stopped activity maintains its state; however, it is not visible to the user and can be killed if the memory situation is critical.

Paused and stopped activities can be terminated due to insufficient memory by asking the activity to finish. When the paused or stopped activity is re-opened, it must be created all over.

Activities are part of an activity lifecycle, in Figure 2.5, the state of the activity can be vaguely categorized into:

- **Entire Lifetime**: of an activity occurs between the calls to `OnCreate()` and the call to `OnDestroy()`. The activity sets the states (e.g., defining the layout) in `OnCreate()`, and release remaining resources in `OnDestroy()`.
- **Visible Lifetime**: of an activity happens between the calls to `onStart()` and the call to `onStop()`. Within this lifecycle, the user can see and interact with the application. Any resources that impact

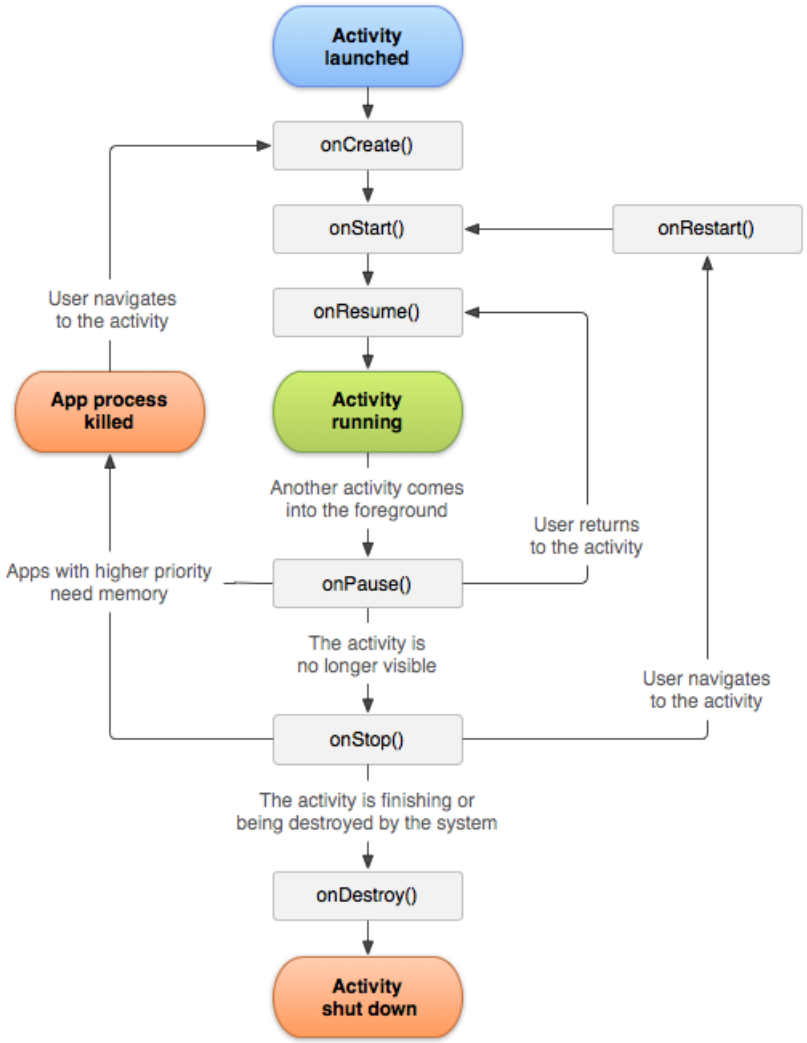

Figure 2.5: Android activity lifecycle [2].

or affect the application occurs between these methods. As activities can alternative between state, the system might call these methods multiple times during the lifecycle of the activity.

- **Foreground Lifetime**: of an activity occurs between the calls to `onResume()` and `onPause()`. The activity is on top of the stack and has user input focus. An activity can frequently transition in this state; therefore, ensuring that the code in these methods is lightweight in order to prevent the user from waiting.

Fragment

A fragment represents a behavior or is a part of a user interface that can be placed in an activity [22]. Fragment allows for reuse of user interface or behavior across applications and can be combined to build a multi-pane user interface inside an activity. The fragment allows for more

flexibility around the user interface, by allowing activities to comprise of multiple fragments which will have their own layout, events, and lifecycles. The lifecycle of a fragment is quite similar to the activity lifecycle; with extended states for fragment attachment/detachment, fragment view creation/destruction, and host activity creation/destruction. A fragment is coherent with its host activity, and the state of the fragment is affected by the state of the host activity.

2.2.2.2 Service

Service is a component that runs in the background to perform long-running tasks [47]. The application or other applications can start a service which remains in the background even if the user switches applications. In contrast, activities are not able to continue if the user switches to another application. Also, a service can bind with a component to interact or perform inter-process communication (IPC). To summarize, a service has two forms:

- **Started**: A component can call the `startService()` method on a service, such that the service can run in the background.
- **Bound**: A component can call the `bindService()` method on a service, which in return will offer a client-server interface to perform operations (e.g., sending requests, or retrieving results) across processes with inter-process communication (IPC). Multiple components can bind to a service, and the last component to `unbind` will destroy the service.

2.2.2.3 Broadcast Receiver

A broadcast receiver is a component that receives broadcast announcements mostly originating from the system (e.g., the screen turned off, the battery is low, or a picture was captured). Applications can subscribe to messages, and the `BroadcastReceiver` can address and process the messages accordingly. Applications can also initiate broadcasts, and the data is delivered as an `Intent` object. A `BroadcastReceiver` can be registered in the activity of the application (with `IntentFilter`), or inside of the manifest file.

2.2.2.4 Content Providers

Content providers manage access to a set of structured data and provide a mechanism to encapsulate and secure the data [15]. Content providers is an interface which enables one process to connect its data with another process. Also, in order to copy and paste complex data or files between applications, a content provider is required. For instance, to share a file

across a media (e.g., mail), a `FileProvider` (subclass of `ContentProvider`) is needed to facilitate a secure sharing of files [21].

2.2.3 Process and Threads

The Android system creates a Linux process with a single thread of execution for an application on launch [44]. All components (i.e., activity, service, broadcast receiver, and providers) run in the same process and thread (called the *main* thread) unless the developer arranges for components to run in a separate process. A process can also have additional threads for processing.

When the memory on the device runs low and demanded by processes which are serving the user, Android might kill low priority processes. Android decides to kill the process based on priority; the process hierarchy consists of five levels (the lowest priority number is the most important and is killed last):

1. **Foreground Process**: is a process that is required by the user to interact and function with the application. A foreground process is categorized as: activity that the user interacts with, service that is bound to an interacting activity, service that is running in the `foreground` (with `startForeground()`), service that is executing on of the lifecycle callbacks, and broadcast receivers executing `onReceive()` method.
2. **Visible Process**: is a process without foreground components, but affect the user interactions. A visible process is when a foreground process takes control (however, the visible process can be seen behind it), and a service that is bound to a visible (or foreground) activity.
3. **Service Process**: is a process that executes work which is not displayed to the user (e.g., playing music or downloading data), and are started with the `startService()` method.
4. **Background Process**: is a process that holds information of paused activities. This process state has no impact on the user experience, and these processes are kept in an LRU (least recently used) in order to refrain killing the activity that the user used last. The state of the process can be saved if the lifecycle method in an activity is implemented correctly, to ensure seamless user experience.
5. **Empty Process**: is a process that does not hold any active application components; however, are kept alive for caching and faster startup time for components that need to be executed.

Threads

The main thread is responsible for dispatching events to the user interface widget and drawing events. Also, the thread interacts with the application components from the Android UI toolkit; the main thread is also called

UI thread. System calls to other components are dispatched from the main thread, and components that run in the same process are instantiated from the main thread. Intensive work (such as long-running operations as network access or database queries) in response to user interaction, can lead to blocking of the user interface. As a consequence, the user can find the application to hang and might decide to quit or uninstall the application.

Additionally, the tool kit to update the user interface in Android is not thread-safe; therefore, enforcing the rules of (1) not to block the UI thread; and (2) not to access the Android UI toolkit outside the UI thread. In order to run long-running or blocking operations, one can spawn a new thread or Android provides several options: `runOnUiThread`, `postDelay`, and `AsyncTask` (perform an asynchronous task in a worker thread, and publishes the results on the UI thread).

2.2.4 Inter-Process Communication (IPC)

Inter-process communication is a mechanism to perform remote procedure calls (RPC) to application components that are executed remotely (in another process), with results returned to the caller. To perform IPC, the caller has to bind to a remote `service` (using `bindService()`). Upon binding to a remote service, a proxy used for communication with the remote service is returned. The proxy decomposes the method calls, and the Binder framework takes the methods and transfers them to the remote process [10]. Android offers a language to enable IPC; called Android Interface Definition Language (AIDL) [6].

Besides AIDL, one can use `Intent` to pass messages across processes. The intent is a messaging object used to request action from other application components [28]. There are two types of intents: (1) `explicit intents`: used to start a component in the application, by supplying the application package name or component class; and (2) `implicit intents`: declare a general action to perform, which enables other applications to handle it. The main uses cases of intent are to starting an activity, starting a service, and delivering a broadcast.

2.2.5 Data and File Storage

Android provides options to store application data on the device, depending on space requirement, data type, and whether the data should be accessible to other application or private to the application [18]. There are four distinctive data storage options, depending on the requirement of data that is being stored:

- **Interal File Storage**: The system provides a directory on the file system for the application to store and access files. By default, the

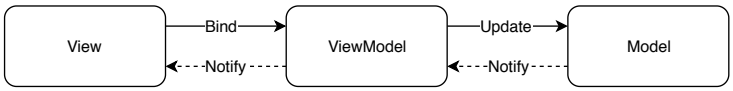

Figure 2.6: Structure of the Model-View-ViewModel architectural pattern.

files saved in this directory is private to the application. Also, files stored in the internal storage are removed on uninstallation of the application; therefore, storing persistent data that are expected to be on the device regardless of the application removal, should not be using internal file storage. In addition, internal file storage allows for caching, which enables temporarily data storage (that do not require to be persistent).

- **External File Storage**: External file storage enables storing of files in a space where users can mount to a computer as an external storage device (or to physical removable storage, such as SD card). Files stored into external storage makes it possible to transfer files on a computer over USB. Files stored in external file storage enables other applications to access the data, and the data remain available after application uninstallation (unless specified that the storage is application-specific).
- **Shared Preferences**: For storing small and unstructured data, `SharedPreference` enables API to read and write persistent key-value pairs of primitive data types (e.g., booleans, floats, ints, longs, and strings). The storage location is specified by uniquely identifying the name, and the data is stored into an XML file. Also, the data stored remain persistent (even after application termination).
- **Databases**: Android provides support for SQLite databases, which is a relational database management system embedded in the system. The access to the database is private to the application, and accessing the database can be done with the `Room persistence library`. The Room library provides an abstract layer over SQLite APIs.

2.2.6 Architecture Patterns

The architectural pattern principle enhances the separation of *graphical user interface* logic from the *oparting system* interactions [27]. The Model-View-ViewModel (hereafter: MVVM) is an architectural pattern which is well-integrated in Android. It has three components that constitute the principle:

- **Model**: represents the data and the business logic of the application.
- **ViewModel**: interacts with the model, and manages the state of the view.
- **View**: handles and manages the user interface of the application.

In Figure 2.6, the interactions amongst the components are illustrated. The connection between the `View` and `ViewModel` occurs over a data binding connection, which enables the view to change automatically based on changes to the binding of the subscribed data [36]. In Android, the `LiveData` is an observable data holder that enables data binding, which allows components to observe for data changes. `LiveData` respects the lifecycle of the application components (e.g., activities, fragments, or services), ensuring the `LiveData` only updates the components that are in an active lifecycle state [34]. Moreover, `Android Room` provides a set of components to facilitate the structure of the model component [45]. More specifically, it models a database and the entities (which are the tables in the database).

2.2.7 Power Management

Battery life is a concern to the user, as the battery capacity is significantly limited on devices [43]. Android has features to extend battery life by optimizing the behavior of the application and the device, and provides several techniques to improve battery life:

- **App Restrictions**: Users can choose to restrict applications (e.g., applications cannot use the network in the background, and foreground services). The application that is restricted, functions as expected when the user launches the application; however, are restricted to do background tasks.

- **App Standby**: An application can be put into standby mode if a user is not actively using it, resulting in the application background activity and jobs are postponed. An application is in standby mode if there is no process in the foreground, a notification is being viewed by the user, and not explicitly being used by the user.

- **Doze**: When a device is unused for a long period, applications are delayed to do background activities and tasks. The doze-mode enters maintenance window to complete pending work, and then resume sleep for a longer period of time. This cycles through until a maximum of sleep time are reached. Some applications want to keep the device running to perform long-running tasks (e.g., collecting data), and `WakeLocks` enables this. `WakeLocks` allows the application to perform activities and tasks, even while the screen is turned off [30].

- **Exemptions**: Another way of keeping an application awake, is to exempting applications from being forced to `Doze` or in `App Standby`. The exempted applications are listed in the settings of the device, and users can manually choose the application to exempt. Consequently, exempted applications might overconsume the battery of the device.

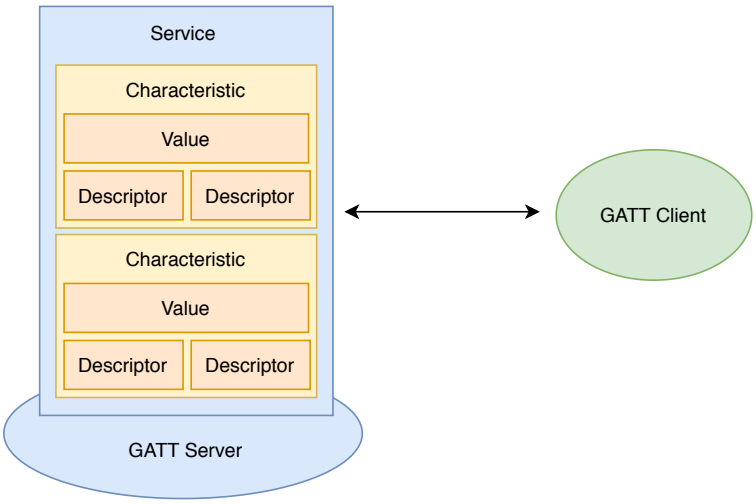

Figure 2.7: The structure of BlueTooth Low Energy; illustrating the GATT Server with a service containing many characteristics and its value, and the GATT client connecting with the GATT server.

2.2.8 Bluetooth Low Energy

Android supports for Bluetooth Low Energy (BLE), which is designed to provide lower power consumption on data transmission (in contrast to classic Bluetooth) [11]. BLE allows Android applications to communicate with sensors or devices (e.g., heart rate sensor, and fitness devices), that has stricter power requirements. Sensors that utilize BLE are designed to last for a longer period (e.g., weeks or months before needing to charge or replace battery). The protocol of BLE is optimized for a small burst of data exchange, and terms and concepts that form a BLE can be characterized as:

- **Generic Attribute Profile (GATT)**: As of now, all Low Energy applications are based on GATT. GATT is a general specification for sending and receiving a burst of data (known as *attributes*) over a BLE link.
- **Attribute Protocol (ATT)**: GATT utilizes the Attribut Protocol, which uses a few bytes as possible to be uniquely identified by a Universal Unique Identifier (UUID). A UUID is a standardized format to identify information.
- **Characteristic**: Contains a single value and descriptors that describe the value of the characteristics (i.e., can be seen as a type).
- **Descriptor**: Are defined attributes that describe a characteristic value (e.g., specify a human-readable description, a range of acceptable values, or a unit of measure).
- **Service**: Is a collection of characteristics (e.g., a service is *Heart Rate Monitor* which includes a characteristic of *heart rate measurement*).

In order to enable BLE, facilitating for a GATT server and GATT client is required. Either the sensor or the device takes the role of being a server or a client. However, the GATT server offers a set of services (i.e., features), where each service has a set of characteristics. Moreover, the GATT client can subscribe and read from the services the GATT server provides.

Chapter 3

Related Work

The detection and analysis of sleep-related breathing disorders on a mobile device has been a research topic and concert for some time. Various techniques and methods have been applied to detect sleep-related breathing disorders on mobile phone with the use of built-in or external sensors. In this chapter, we survey some of the research conducted in this field.

Nandakumar, Gollakota, and Watson [40] present a contactless solution for detecting sleep apnea events on smartphones. The goal behind the research is to detect sleep apnea events without any sensor on the human body. They achieved this by transforming the phone into an active sonar system that emits frequency-modulated sound signals and observes for the reflections. Based on the experiments, the system operates efficiently at distances of up to a meter, also while the subject is under a blanket. They performed a clinical study with 37 patients and concluded that the system managed to accurately compute the central, obstructive, and hypopnea events (with a correlation coefficient of 0.99, 0.98, and 0.95).

Alqassim *et al.* [3] designed and implemented a mobile application (for Windows and Android) to monitor and detect symptoms of sleep apnea using built-in sensors in the smartphone. The purpose of the application is to make users aware of whether they have sleep apnea before they continue with a more expensive and advanced sleep test. They achieved this by measuring the breathing patterns and movements patterns based on the built-in microphone and accelerometer in the smartphone. The system instructs the user to place the smartphone on its arm, abdomen, or near the bed during recording. The data is collected on the smartphone and sent to a central server in the cloud, where authorized doctors can review the samples. To summarize, the system tries to monitor sleep apnea in the aspect of motion and voice recorder, in order to detect sleep apnea on a smartphone.

Penzel *et al.* [41] investigates the challenges and develop a system associated with insufficient conventional sleep laboratories and their

expensive and time-consuming polysomnographic diagnostics as early as in 1989. The purpose of the system is to make the diagnosis of sleep-related breathing disorders available at any hospital; the system was placed in a wooden case with wheels to be moved between bedside locations. They developed a circuit board that has support for various sensor to evaluate breathing, ECG, blood gases, and the state of sleep. During record, the samples from the sensors were compressed to a resolution of one value per second per parameter (e.g., mean value of respiratory frequency, ventilated volume, actigraph activity, and EOG activity) and stored on a personal computer. After the recording, manual evaluation can be carried out using print-outs, as well as reviewing the data on a screen. Thus, the software of the system facilitates recording and reviewing of the data, also basic evaluation and analysis.

3.1 Summary & Conclusion

To summarize, Nandakumar, Gollakota, and Watson created an application to collect samples through a contactless solution, which quite accurately measures central, obstructive, and hypopnea events. Alqassim *et al.* developed a mobile application to sample breathing patterns and movement patterns with the use of the built-in microphone and accelerometer in smartphones. Finally, Penzel *et al.* built a mobile system to record and analyze sleep-related breathing disorders with technology that was advanced at the time.

To conclude this chapter, the developing of a system that records, analyzes, and detects sleep-related breathing disorders (e.g., obstructive sleep apnea) proposes a solution to aid researchers/doctors in analyzing and detecting sleep apnea in patients. Based on the related work in this chapter, we observe that there are distinguishable techniques and methods for the detection and analyzing of sleep-related breathing disorders through the use of mobile systems. Considering this, the future might introduce more techniques and methods that improve the analysis and detection of sleep apnea. Thus, developing a system that is extensible and modular to new techniques and methods should be considered. Therefore, we look into the opportunities of creating an application that enables the support for third-party modules, these modules extend the functionality of the application or provide more enrichment to the collected data on the patients' device.

Part II

Design and Implementation

Chapter 4

Analysis and High-Level Design

It is the motivation of this thesis to enable early detection of sleep-related disorders with the aid of an Android device and low-cost sensors, and to provide further analysis and evaluation in sleep- and breath-related patterns. We developed an application, called *Nidra*, which attempts to record, share, and analyze data collected from external sensors, all on a mobile device. Also, Nidra acts as a platform for modules to enrich the data, thus extending the functionality of the application.

The motive behind this application is to provide an interface for patients to record breathing data from home and to aid researchers and doctors with analysis of sleep-related breathing disorders (e.g., obstructive sleep apnea) in a patient. An overview of the CESAR project is found in Figure 2.1, separating the project into three parts, in which the application part is what we intend to extend the project with, by introducing Nidra. As for now, Nidra consists of three main functionalities, each related to the requirements defined in the problem statement:

1. The application must provide an interface for the patient to (i) record physiological signals (e.g., breathing data); (ii) present the results; and (iii) share the results.

2. The application must ensure that upon sensor disconnections, the connectivity is reinstated to minimize the data loss and its effects on the data analysis.

3. The application must provide an interface for the developers to create modules that integrate with the application.

This chapter gives a detailed look at the design of Nidra, including the tasks which are separated into concerns that combined constitute the structure of the application—followed up the structure of the data in the application.

4.1 Requirement Analysis

4.1.1 Stakeholders

McGrath and Whitty [38] describe the term stakeholders as those persons or organizations that have, or claim an interest in the project. They distinguish stakeholders into four categories: (1) *contributing (primary) stakeholders* participate in developing and sustaining the project; (2) *observer (secondary) stakeholders* affect or influence the project; (3) *end-users (tertiary stakeholder)* interact and uses the output of the application; and (4) *invested stakeholders* have control of the project. In Nidra, three stakeholders affect the application, and each is categorized respectively:

- `Patients` are identified as an end-user—they interact with the output of the application.
- `Researchers/Doctors` are identified as an observer stakeholder—they might not use the application itself; however, they might use the data obtained from the patients' recordings for further analysis. Also, request more functionality in the application.
- `Developers` are identified as a contributor stakeholder—they maintain the application from bugs. Additionally, they can contribute to developing modules that extend the functionality of the application.

4.1.2 Resource Efficiency

The application is designed for the use on a mobile device; modern mobile devices are empowered with multi-core processors, a sufficient amount of RAM, and a variety of sensors. However, the battery capacity is restrictive and based on usage. The device may only last for one day before a charge, due to the size of the battery capacity [33]. The average battery capacity of a mobile device is approximately 2000 mAh on budget devices and approximately 3000 mAh on high-end devices [20]. The application should be able to run at least 7 hours without any power supply. Also, the device should be capable of handling various sensor connections simultaneously. Therefore, the application should be designed to be resource-efficient by utilizing the least amount of battery resources during a recording. Also, ensure a sufficient amount of power on the device before starting a recording session.

4.1.3 Security and Privacy

The proposed use of the application is to monitor the sleeping patterns of a patient. The application manages and stores personal- and health-related data about the patient. As a precaution, the application should incorporate the CIA triad, which stresses data confidentiality, integrity, and availability

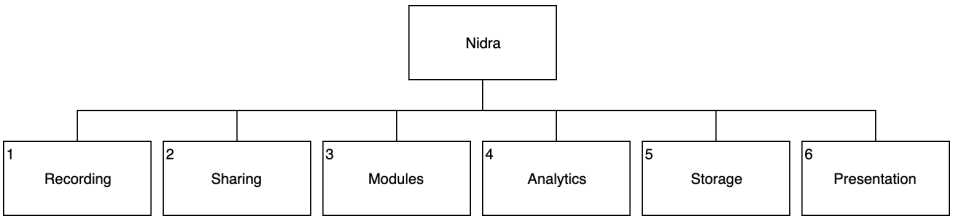

Figure 4.1: The individual tasks that outline the application: (1) recording, (2) sharing, (3) modules, (4) analytics, (5) storage, and (6) presentation.

[52]. Any unauthorized access to the data, data leaks and confidentiality should be appropriately managed on the device. Sharing the data across application or with researchers/doctors should be granted with the consent from the patient. Besides, a mobile device can be connected to the Internet, which makes it vulnerable to attacks. Also, other installed application on the device can manipulate the access to the data. Therefore, revising the security policy defined by Android [46] should be incentivized.

4.2 High-Level Design

4.2.1 Task Analysis

Task analysis is a methodology to facilitate the design of complex systems. Hierarchical task analysis (HTA) is an underlying technique that analyzes and decomposes complex tasks such as planning, diagnosis, and decision making into specific subtasks [16]. In Figure 4.1, an illustration of the tasks of the application is presented. This section introduces these tasks, which are an integral part of the development of the application.

Recording

A *recording* is a process of collecting and storing physiological signals (e.g., breathing data) from sensors over an extended period (e.g., overnight). To enable a recording, we need to establish connections to available sensors, collect samples from the sensors, and store the samples on the device. A *sensor* is a device that transforms analog signals from the real world into digital signals. The digital signals are transmittable over Link Layer technologies (e.g., BlueTooth), and the communication between a sensor and device occurs based on the protocols the sensor supports. A *sample* is a single sensor reading containing data and metadata, such as time and the physiological data. During the recording session, ensuring that the sensors and the devices maintain connectivity is important, such that the record contains more meaningful data upon analysis. Once a recording session has terminated, a *record* with metadata about the recording session is stored, alongside the samples.

Sharing

Sharing is a mechanism to export and import records across applications. *Exporting* consists of bundling one or more records with correlated samples into a transmittable format and transferring the bundled records over a media (e.g., mail). *Importing*, on the other hand, consists of locating the bundled records on the device, parsing the content and storing it on the device. The sharing mechanism allows the patients to send their records to researchers/doctors.

Module

A *module* is an independent application that is installed and launched in Nidra (hereafter: application), to provide extended functionality and data enrichment. A module does not necessarily interact with the application. However, it utilizes the data (e.g., records). For example, a module could be using the records to feed a machine-learning algorithm to predict obstructive sleep apnea. Installing a module is achieved by locating the module-application on the device, and storing the reference in the application. Due to limitations in Android, the module-application cannot be executed within the application. Therefore, the module-application is a standalone Android application. Furthermore, the development of the module-application is independent from the application.

Analytics

Analytics is the visualization and interpretation of patterns in the records. The application facilitates the recording of breathing data, which aids in the detection and analysis of sleep-related breathing disorder. There are various analytical methods, ranging from graphs to advanced machine learning algorithms, and incorporating a simple time series plot can indirectly aid in the analysis. For example, plotting a time series graph where the breathing data are on the Y-axis and the time on X-axis, provides a graphical representation of the data that can be further analyzed within the application.

Storage

Storage is the objective of achieving persistent data; data remain available after application termination. To enable storage, we use a database for a collection of related data that is easily accessed, managed, and updated. The database should be able to store records, samples, modules, and biometrical data related to the user (i.e., gender, age, height, and weight). Structuring a database that is reliable, efficient, and secure is a crucial part of achieving persistent storage. Android provides several options to enable storage on the device (e.g., internal storage and database).

Presentation

Presentation is the concept of exhibiting the functionality of the application to the user. A user interface (UI) is the part of the system that facilitates interaction between the user and the system. In Nidra, determining the

screen layout, color palette, interactions, and feedback on actions is part of the development of a user interface.

4.3 Seperation of Concerns

Separation of concern is a paradigm that classifies an application into concerns at a conceptual and implementational level. It is beneficial for reducing complexity, improving understandability, and increasing reusability [32]. The concerns in this thesis are the individual tasks defined in the task analysis. Each concern is conceptualized with a graph of *components*, the functionality of each component when combined constitutes a structure, that is derived based on iterative work and research. This section proposes a solution for the structure of each concern, by analyzing and decomposing the tasks defined in task analysis into components.

4.3.1 Recording

The structure of a recording is restrictive in terms of arranging the components due to the design of the CESAR project. That is, the recording structure is based on the interface (e.g., establishment and receiving data) facilitated for the sensor sources by the project. Besides this, the structure can be extended with components that monitor the data acquisition and the connectivity with the sensor and the device. In Figure 4.2, a proposition of the structure for recording with components and their dependencies is illustrated, and the components are described as:

1.1 `Sensor Discover`: Find all eligible sensors that can enable a recording.

1.1.1 `Select Sensors`: Select a preferable sensor source for data acquisition.

1.1.2 `All Sensors`: Select all available sensors source for data acquisition.

1.2 `Sensor Initialization`: Establish and initialize a connection with the selected sensor source. Occasionally a sensor might use some time to connect, or unforeseen occurrence is hindering the initialization of the sensor. Therefore, halting the sensor initalization or actively checking for sensor initalization is important.

1.3 `Sensor Connectivity Setup`: Establish a connection between the application and the sensor source using the protocols the sensor sources provides (e.g., BlueTooth). All data exchange will occur over the established interface.

1.4 `Connection State`: Is the state that determines whether the recording has started (ongoing) or stopped (finalized).

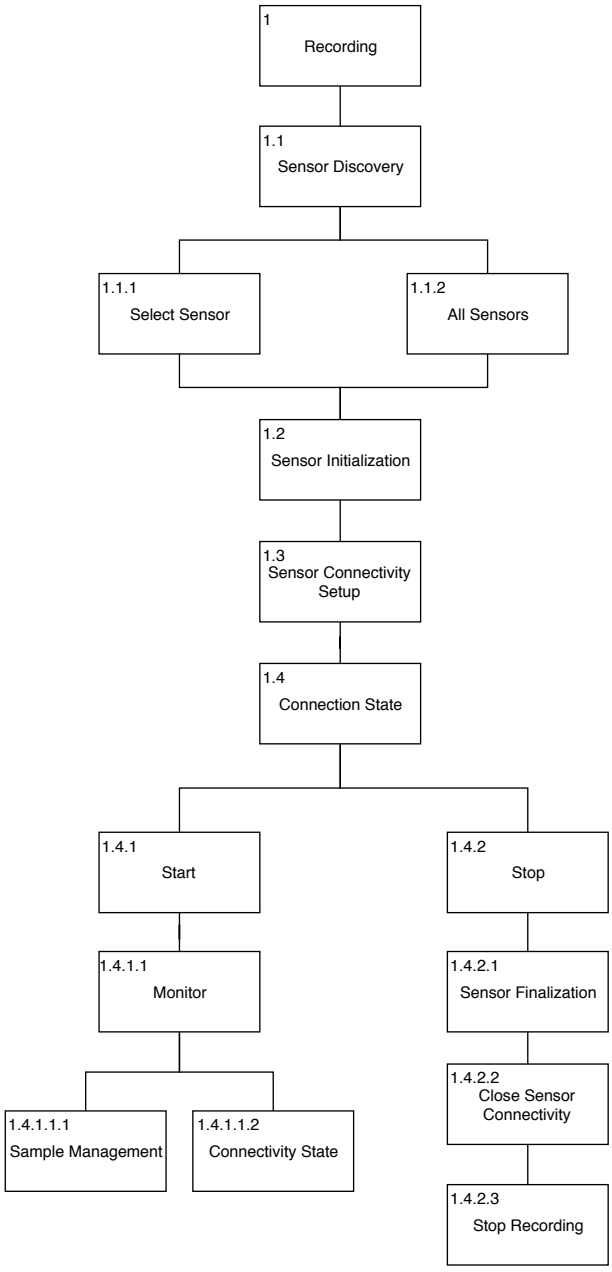

Figure 4.2: A design proposal for the structure of the recording concern.

1.4.1 `Start`: Notify the sensors to begin collecting data, and the application should display that a recording has started to the user. Also, start a timer to display time elapsed on the current recording.

1.4.1.1 `Monitor`: Is a mechanism to handle the connectivty state and the incoming samples. It is actively listening to the interface for new data from the sensors, and forwarding the data to the sample management component.

1.4.1.1.1 `Sample Management`: Handles a single sample from a sensor by parsing the content according to the payload of the data tuple from the sensor.

1.4.1.1.2 `Connectivity State`: Makes sure that the connectivity between the sensor and the device is maintained. Further described in Section 4.3.1.1.

1.4.2 `Stop`: Stop the recording timer and proceed to display results.

1.4.2.1 `Sensor Finalization`: Notify the sensor to stop sampling data, and close establishment.

1.4.2.2 `Close Sensor Connectivity`: Close the interface connections between the application and the connected sensors.

1.4.2.3 `Stop Recording`: Once all connections are closed, add additional information to the recording (e.g., title, description, rating). In the end, the recording has concluded and it is stored on the mobile device.

4.3.1.1 Connectivity State Component

Connectivity state is a component that monitors for unexpected sensor disconnections or disruptions. Unexpected behavior can occur due to anomalies in the sensor, or the sensor being out of reach from the device for a brief moment. A naive solution would be to ignore the connectivity state component, and assuming the sensors are connected to the device indefinitely. However, upon disconnections or disruptions, the recording would be missing samples, resulting the record has less meaningful data. This component enables the application to reconnect with the sensor based on a time interval, resulting in more accurate and meaningful record. The following design questions for this component are (1) should the connectivity state component, which implements a time interval that tries to reconnect with the sensor, be implemented in the sensor wrapper, or should it be in the proposed recording structure?; and (2) should the interval between sample arrival be a fixed time or a dynamical time?

1. To achieve a mechanism of reconnecting with the sensor on unexpected disconnects or disruptions, establishing a time interval that monitors for sample arrivals within a time frame (e.g., every 10 seconds) is required. Incorporating the time interval in the sensor wrapper

reduces the complexity of Nidra. However, it introduces extra complexity to the sensor wrappers. A sensor wrapper has to distinguish actual disconnects from unexpected disconnects. Although, by extending the functionality of sensor wrapper by implementing a state that indicates whether a recording is undergoing or stopped solves the problem. All future sensor wrappers would then have to implement the proposed solution, resulting in a complicated and time-consuming sensor wrappers development. While implementing the proposed solution in the sensor wrappers is possible, extending the recording structure with the logic in Nidra would be more meaningful and time-saving. In our design, we implement the connectivity state in the structure for recording.

2. A time interval triggers an event every specified time frame. A time frame can be in a fixed size (e.g., every 10 seconds) or a dynamical size (e.g., start with 10 seconds, then incrementally increase the frame by X seconds). Implementing a fixed time frame increases the stress on put on the sensor, whereas a dynamical time might miss samples if the time frame is significant. Depending on how critical the recording is, a suitable solution for the time frame should be configurable. Also, limiting the number of attempts made to reconnect should be considered, due to actively reconnecting to a sensor that presumably is dead or completely out of reach is unnecessary. Thus, stopping the recording once a limited number of attempts has been reached. In our design, we implemented a dynamical time and limited the number of attempts to 10.

4.3.2 Sharing

Sharing is separated into two concerns: export and import. The scope of exporting in Nidra is to select desired records, format and bundle the records into a transmittable file, and sendting the bundle over a media (e.g., mail). The scope of importing is to locate the file on the device, parse the content based on the format, and store it on the device. In Figure 4.3, a proposition of the structure for sharing with components and their dependencies is illustrated, and the components are described as:

2.1 `Import`: Is a mechanism that locates a file, parses the data, and stores it on the device.

2.1.1 `Locate File`: To enable this, the user has to download the file on the device. Then, locate the file on the device by using an interface to browse downloaded files. An interface can be developed; however, using the interface Android provides to locate downloaded files is a more straightforward solution.

2.1.2 `Parse Formated Content`: Parse the content of the file accordingly to the data format discussed in Section 4.4.1.

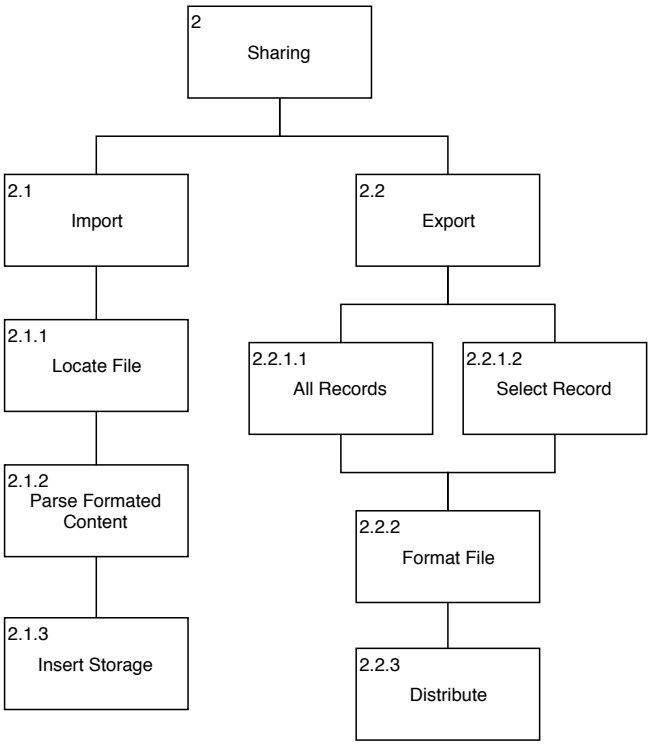

Figure 4.3: A design proposal for the structure of the sharing concern.

2.1.3 `Insert Storage`: Retrieve the necessary data from the parsed file, to store on the device without overriding existing data.

2.2 `Export`: Is a mechanism that selects all or a specific record, transforms the record into a formatted file (see Section 4.4.1), and transmits the file across application (e.g., sending records from the patient's installed application to the researcher/doctor's installed application with the use of an e-mail application).

2.2.1.1 `All Records`: Export all of the records on the device.

2.2.1.2 `Select Record`: Pick one specific record to export.

2.2.2 `Format File`: When a preferred format for the records is selected, bundling the data into a formatted (see Section 4.4.1) file for transmission. It is essential to identify the name of the file uniquely to prevent duplicates and overrides of data. For instance, identifying the name of the file with the device identification appended with the time of exporting.

2.2.3 `Distribute`: Send the file over a media, e.g., e-mail (see Section 4.3.2.1).

4.3.2.1 Distribute Component

The distribute component uses the formatted file and transfer it to other instances of Nidra using a media (e.g., e-mail). There are two distinctive methods to perform this task which is efficient and practical: (1) establish a web-server with logic to handle users and sharing data with the desired recipient, and implementing an interface to retrieve the file within the application; and (2) using the interface provided by Android to share files across applications (e.g., e-mail).

While the first option might be favorable in terms of practicality, this solution introduces additional concerns (e.g., the privacy matters of storing user data on a server) which is out of the scope for this project. For this reason, using the interface provided by Android is a reasonable solution. The user of the application can utilize the Android interface for sharing files over installed applications (e.g., the e-mail is a flexible media to transfer the file, and the user can specify the recipients accordingly).

4.3.3 Modules

Modules are independent applications that provide extended functionality or data enrichment to Nidra. The components for locating and launching a module is given by the Android design; however, the component for data exchange between a module and Nidra can be independently designed. In Figure 4.4, a proposition of the structure for modules with components and their dependencies is illustrated, and the components are described as:

3.1.1 `Install Modules`: Is the process of locating the application on the device, and storing the reference of the application package name in the storage.

3.1.2 `Locate Module`: Retrieve the list of stored modules, and display the installed modules to the user.

3.2 `Launch Module`: Get the application location stored in the module, and launch the application with the use of Android Intent.

3.3 `Data Exchange`: Enrich the module with data from the application (see Section 4.3.3.1).

4.3.3.1 Data Exchange Component

The data exchange component facilitates the transportation of data between Nidra and a module. As of now, the data is records and corresponding samples, which is formatted (see Section 4.4.1) accordingly. The two distinct methods to exchange data between a module and Nidra are (1) formatting all of the data and bundling it into the launch of the mod-

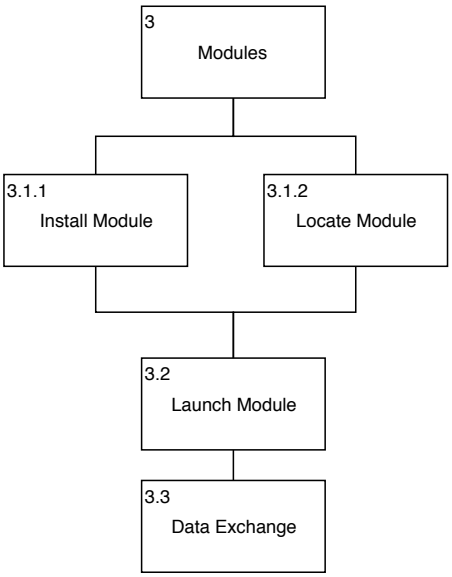

Figure 4.4: A design proposal for the structure of the modules concern.

ule, and (2) establishing a communication link for bi-directional requests between Nidra and the module.

Android provides an interface to attach extra data on activity launch. The first solution is, therefore, convenient and efficient; all of the data is formatted and bundled into the launch. However, once Nidra has launched the module, there are no ways of transmitting new data besides relaunching the module. For this reason, the second option allows for continuous data flow by establishing a communication link with IPC between the applications. The data exchange between Nidra and modules can then be bidirectional; the module can request desired data any time, and Nidra can collect reports and results generated by the module.

One could argue that new records are not obtained while managing and using a module. However, there might be future modules that do a real-time analysis of a recording; as such, require an interface for continuous data flow. For the simplicity of our design, we use the first option of bundling all of the data and sending it on launch.

4.3.4 Analytics

Analytics uses techniques and methods to gain valuable knowledge from data. Nidra provides a simple time-series plot for illustration of the data. However, other techniques can be in incorporated. In essence, the facilitation of modules in the application enables the development opportunities for advanced analytics of the data. In Figure 4.5, the structure of analytics is presented with components and their dependencies:

4.1.1 `Select Record`: Select one of the records on the device.

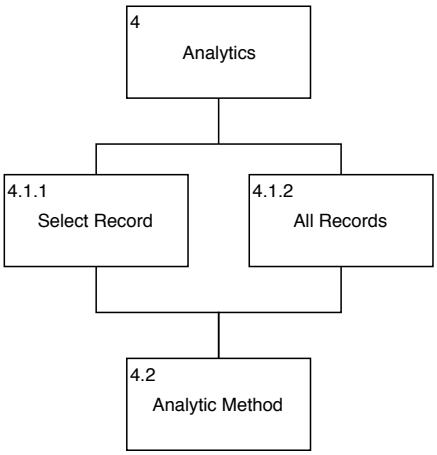

Figure 4.5: A design proposal for the structure of the analytics concern.

4.1.2 `All Records`: Select all of the records on the device.

4.2 `Analytic Method`: Apply an analytical method on the data provided in Nidra (see Section 4.3.4.1).

4.3.4.1 Analytic Method Component

The analytic method component uses the records on the device for representation or analysis. Graphical and non-graphical are two techniques for representing data. Graphical techniques visualize the data in a graph that enables analysis in various ways. A few graphical techniques are diagrams, charts, and time series. Non-graphical techniques, better known as statistical data tables, represent the data in tabular format. This provides a measurement of two or more values at a time [24]. More advanced techniques to analyze the data are to use machine learning. Machine learning is concerned with developing data-driven algorithms, which can learn from observations without explicit instructions. For example, using recurrent neural networks (e.g., RNN, LSTM) or regression models (e.g., ARIMA), can be used to predict the sleeping patterns [23].

In Nidra, a time series graph is used to represent the data of a record. The time series graph represents the respiration data on the Y-axis and the time on the X-axis. Essentially, the facilitation of modules in the application is designed to enable advanced techniques to predict, analyze, and interpret the data acquisition. Therefore, in Nidra, the analytic methods are limited; however, the modules enable developers to construct any method they desire.

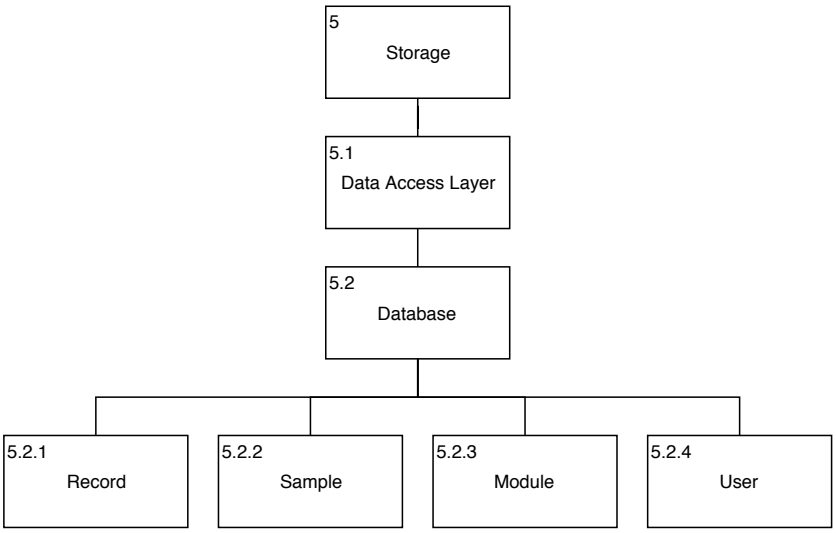

Figure 4.6: A design proposal for the structure of the storage concern.

4.3.5 Storage

Storage is the objective of achieving persistent data; data remain available after application termination. The data is characterized into four data entities (i.e., record, sample, module, and user) that contain individual properties. The components in the storage structure are constructed to be extensible and scalable in terms of future data, restructure of data, and removal of data. In Figure 4.6, a proposition of the structure for storage with components and their dependencies is illustrated, and the components are described as:

5.1 `Data Access Layer`: It exposes specific operations (such as insertion of a record) without revealing the database logic. This component is also known as a data access object (DAO), that provides an abstract interface to a database. The advantage of this interface is to have a single entry point for each database operation and to extend and modify the operation for future data easily.

5.2 `Database`: Is the storage of all the data (see Section 4.3.5.1).

5.2.1 `Record`: Is a table in the database, which contains fields appropriately to the record structure. A record contains meta-data about a recording (e.g., name, recording time, user). The design decision and an example of a recording record are illustrated in Section 4.4.2.1.

5.2.2 `Sample`: Is a table in the database, which contains fields appropriately to a single sample from a sensor. A sample contains data received from a sensor during a recording. The design decision and an example of a sample record are illustrated in Section 4.4.2.2.

5.2.3 `Module`: Is a table in the database, which contains fields appropriately to the module structure. A module contains the name of the

module-application and a reference to the application package. The design decision and an example of a module record are illustrated in Section 4.4.2.3.

5.2.4 `User`: Is an object stored on the device, which contains fields appropriately to the user structure. A user contains the patient's biometrical information (e.g., name, weight, height). The design decision and an example of a user are illustrated in Section 4.4.2.4.

4.3.5.1 Database Component

Android provides several options to store data on the device; depending on space requirement, type of data that needs to be stored, and whether the data should be private or accessible to other applications. Two suitable options for storage are (1) internal file storage: storing files to the internal storage private to the application; and (2) database: Android provides full support for SQLite databases, and the database access is private to the application [19]. Based on the options, the design question for this component is should the data be stored in a flat file database on the internal file storage, or should it be stored in an SQLite database?

Flat files database encode a database model (e.g., table) as a collection of records with no structured relationship, into a plain text or binary file. For instance, each line of text holds on a record of data, and the fields are separable by delimiters (e.g., comma or tabs). Another possibility is to encode the data in a preferable data format, such as JSON or CSV files (see Section 4.4.1). Flat file databases are easy to use and suited for small scale use; however, they provide no type of security, there is redundancy in the data, and integrity problems [37]. Locating a record is made possible by loading the file, and systematically iterating until the desired record is found. Similarly, updating a record and deleting a record. Consequently, the design of flat file databases is for simple and limited storage.

SQLite is a relational database management system, which is embedded and supported in Android. A relational database management system (RDBMS) provides data storage in fields and record, represented as columns (fields) and rows (records), in a table. The advantage is the ability to index records, relations between data stored in tables, and support querying of complex data with a query language (e.g., SQL). Also, RDBMS provides data integrity between transactions, improved security, and backup and recovery controls [37].

While a flat-file database is applicable to store small and unchangeable data, it is not suitable for scalable and invasive data change. The samples acquisition during recording makes it unreasonable to use a flat-file database. In the design of Nidra, SQLite is a preferable solution for storing sample entities. As a result, the record entity is be stored in an SQLite database, in order to associate it with the sample entity (with a foreign key). As for the module entity, it is more convenient to store and

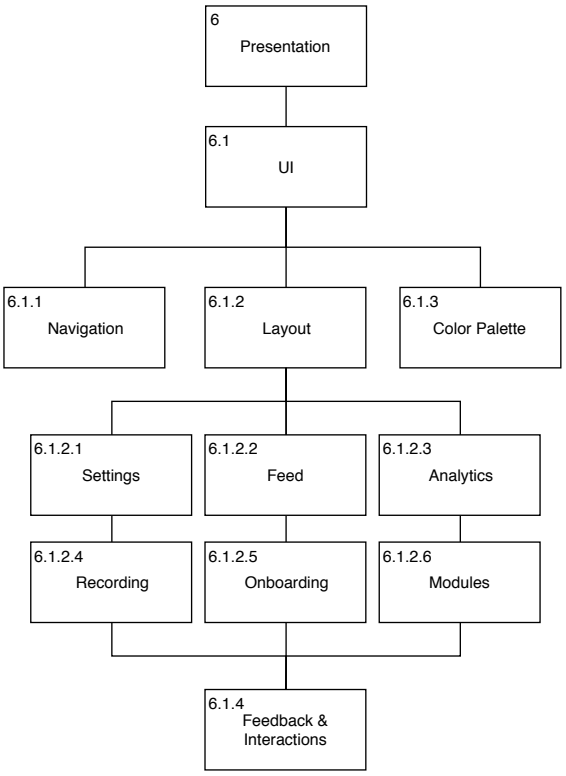

Figure 4.7: A design proposal for the structure of the presentation concern.

update a list of records inside a database; therefore, we use the SQLite database for the module entity. Finally, the user entity is stored in the flat file, that is because there only is one user in the application a time; hence, it is more convenient to use a flat-file (e.g., JSON) to store this type of information.

4.3.6 Presentation

Presentation facilitates the user interface of the application, in terms of visualizing the functionality of the application to the user. The user interface design is guided by the functionality (concerns discussed) in the application and iterative work on the topic. In Figure 4.7, a proposition of the structure for presentation with components and their dependencies is illustrated, and the components are described as:

6.1 `UI`: The user interface where interaction between users and the application occurs.

6.1.1 `Navigation`: The navigation is a menu with options to change the screen (e.g., feed, recording, and module).

6.1.2 `Layout`: Is based on the functionality that is exposed to the user. The layout consists of incorporating the color palette and the screens in the application (the subsequent components).

6.1.3 `Color Palette`: A color palette is a set of colors that persist throughout the application (see Section 4.3.6.1).

6.1.1.1 `Settings`: Is s screen with user details, permissions, and credits, with options to modify permission and user details.

6.1.1.2 `Feed`: Is a list of all records for the user, displayed with details specific to the record to make it distinguishable and easily recognizable.

6.1.1.3 `Analytics`: A interactive time-series graph for a single record.

6.1.1.4 `Recording`: The process of establishing a recording session, in addition to showing the results after a recording session has ended.

6.1.1.5 `Onboarding`: The initial screen displayed to the user, where the user can supply the application with their biometrical data.

6.1.1.6 `Modules`: A list of all installed modules, also an option to add more modules.

6.1.4 `Feedback & Interactions`: Each screen has different feedback and interaction, which should be handled appropriately.

4.3.6.1 Color Palette Component

The color palette is a component that decides the color scheme in the application. In the proposal of a color system in the design guidelines by Google [51], it is essential to pick colors that reflect the style of the application accordingly to: (1) *primary colors*: the most frequently displayed color in the application; (2) *secondary colors*: provides an accent and distinguish color in the application; and (3) *surface, background, error, typography and iconography colors*: colors that reflect the primary and secondary color choices.

Moreover, choosing colors that meet the purpose of the application is critical. Nidra is most likely to be used during the evening and the morning. According to Google [17], a dark color theme reduces luminance emitted by the device screen, which reduces eye strain, while still meeting the minimum color contrast ratios, and conserving battery power. Therefore, in our design, we chose a dark color theme (more specifically, a nuance of dark-blue).

4.4 Data Structure

4.4.1 Data Formats

The data format is a part of the process of serialization, which enables data storage in a file, transmission over the Internet, and reconstruction in a different environment. Serialization is the process of converting the

state of an object into a stream of bytes, which later can be deserialized by rebuilding the stream of bytes to the original object [49]. There are several data serialization formats; however, JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) are the two most common data serialization formats. This section discuss these formats, and in the end, we compare them and choose the format that meets the criteria of being compact, human-readable, and universal.

4.4.1.1 JSON

JSON or JavaScript Object Notation is a light-weight and human-readable format that is commonly used for interchanging data on the web. The format is a text-based solution where the data structure is built on two structures: a collection of name-value pairs (known as objects) and ordered list of values (known as arrays). The JSON format is language-independent and the data structure universally recognized [1, 53]. However, it is limited to a few predefined data types (i.e., string, number, boolean, object, array, and null), and extending the data type has to be done with the preliminary types.

```
1 {
2     "user": {
3         "firstname": "Ola"
4         "lastname": "Nordmann"
5     }
6 }
```

4.4.1.2 XML

XML or eXtensible Markup Language is a simple and flexible format derived from Standard Generalized Markup Language (SGML), developed by the XML Working Group under the World Wide Web Consortium (W3C). An XML document consists of markups called tags, which are containers that describe and organize the enclosed data. The tag starts with < and ends with >; the content is placed between an opening tag and a closing tag. [14, 53] XML provides mechanisms to define custom data types, using existing data types as a starting point, making it extensible for future data.

```
1 <user>
2     <firstname>Ola</firstname>
3     <lastname>Nordmann</lastname>
4 </user>
```

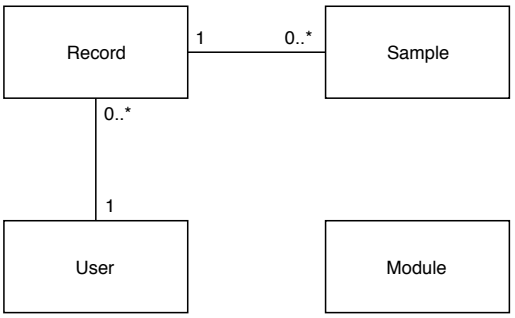

Figure 4.8: The data model and relationship for the entities in the application: a record has zero to many samples, while a sample only can have one record. A record can have one user, while a user can have many records. A module has no relationship with the other entities.

4.4.1.3 Comparing

With the study conducted by Saurabh and D'Souza [53], we compare JSON and XML features and performance. There are apparent differences in the two data formats which affect the overall readability, extensibility, bandwidth performance, and ease of mapping. XML documents are easy to read, while JSON is obscure due to the parenthesis delimiters. XML allows for extended data types, while JSON is limited to a few data types. XML takes more bandwidth due to the metadata overhead, while JSON data is compact and use less amount of bandwidth.

Moreover, a few benchmarks were conducted to measure memory footprint and parsing runtime when serializing and deserializing JSON and XML data. From the conclusion, in terms of memory footprint and parsing runtime, JSON performances better than XML, but at the cost of readability and flexibility. While these format structures are applicable for transmitting data, choosing a format that is compact, human-readable, and a standard format that is extensible and scalable for future data is essential. In our design, we use the JSON format for transmission of the data.

4.4.2 Data Entities

Data entities are objects (e.g., things, persons, or places) that the system models and stores information about. In Section 4.3.5, we introduced four data entities in the application (i.e., record, sample, module, and user). In Figure 4.8, the relations between the data entities are shown. The record entity and sample entity store information about the recording session[1] and are separated into two individual entities in order to reduce data redundancy and improve data integrity. The sample entity has a reference to its record entity so they can be associated with each other. The user

[1] the ongoing process of collecting data from the sensor sources and storing the samples (data packets) in the database.

entity store biometrical information related to the user (i.e., patients). Also, a record entity contains the state of the user's biometrical information at the time of the recording. In other words, the user's biometrical information can change over time (e.g., weight changes); therefore, capturing the exact biometrical information at the time of the recording is essential in the context of detecting sleeping disorders with relation to the biometrical information. A module entity is independent of the other data entities and stores information about the name and the package name of the module-application. The package name is used to locate and launch the module-application.

In the following sections, we present the properties of each data entity in Nidra, and an example of the data structure for each entity.

4.4.2.1 Record Entity

The table for the record entity in the database stores metadata (e.g., elapsed time of recording, number of samples, user's biometrical data) related to a recording session. In Table 4.1, we present the information (fields) the record entity contains, which can be described as:

- `ID`: Unique identification of a record, also a primary key for the entry.
- `Name`: A name of the record to easily recognize the recording.
- `Description`: A summary of the recording session provided by the user. It can be used to briefly describe how the recording session felt (e.g., any abnormalities during the sleep).
- `MonitorTime`: The recording session duration in milliseconds.
- `Rating`: User defined rating on how the sleeping session felt, in a range between 0–5.
- `User`: User's biometrical information encoded into a JSON string format, in order to capture the state of the user at the time of recording.
- `CreatedAt`: Date of creation of the recording in milliseconds (since January 1, 1970, 00:00:00 GMT).
- `UpdatedAt`: Date of update of the recording in milliseconds (since January 1, 1970, 00:00:00 GMT).

id	name	description	monitorTime	rating	user	createdAt	updatedAt
1	Record #1	-	5963088	2.5	{...}	1554406256000	1554406256000

Table 4.1: Example entry in the record table.

4.4.2.2 Sample Entity

The table for the sample entity contains a single sensor reading (data packet) sent from the sensor source. Sample entity are stored separated from a record entity; however, they are linked (foreign key) with their corresponding record entity. In Table 4.2, we present the information (fields) the sample entity contains, which can be described as:

- `ID`: Unique identification of a sample, also a primary key for the entry.
- `RecordID`: An identification to its corresponding record, also a foreign key.
- `ExplicitTS`: Timestamp of sample arrival based on the time in the sensor.
- `ImplicitTS`: Timestamp of sample arrival based on the time on the device.
- `Sample`: Sensor reading contains metadata and data according to Flow sensor.

id	recordId	explicitTS	implicitTS	sample
1	1	1554393086000	1554400286000	Time=0ms, deltaT=100, data=1906,1891,1884,1881,1876,1718,1690

Table 4.2: Example entry in the sample table.

4.4.2.3 Module Entity

The table for the module entity contains metadata of the module added by the user in the application. In Table 4.3, we present the information (fields) the module entity contains, which can be described as:

- `ID`: Unique identification of a module, also a primary key for the entry.
- `Name`: The name of the module-application.
- `PackageName`: The package name of the module-application, such that it can be launched from Nidra.

id	name	packageName
1	OSA Predicter	com.package.osa_predicter

Table 4.3: Example entry in the module table.

4.4.2.4 User Entity

The user of the application is the patient, which provides biometrical data (e.g., weight, height, and age). The biometrical data is part of the application to enrich the record. The record captures the biometrical state

of the user at the time of the recording. In Nidra, the user entity is not a part of the database, but stored as an object on the mobile device. The design decision of this choice is because the application is limited to one user at the time, and it makes it convenient to capture the state of the object of the given time of recording. It is possible to create a table in the database for the user entity, and for each time the user changes the biometrical data insert it is a separate entry in the database. The record could then have a reference to the latest user entry. However, that increases the complexity of the system, and as of now, the design of the application is to store the user's biometrical data into an object on the device.

In the listing below, we present the structure of the user entity. In Nidra, the user entity is stored as an object and structured as a JSON format containing biometrical data related to the user:

- `Name`: A string that contains the name of of the patient.
- `Age`: An integer with the age of the patient.
- `Gender`: A string that contains the gender of the patient.
- `Weight`: An integer with the weight of the patient in kilograms.
- `Height`: An integer with the height of the patient in centimeters.

```
1 {
2     "user": {
3         "name": "Ola Nordmann",
4         "age": 50,
5         "gender": "Male",
6         "height": 180,
7         "weight: 60
8     }
9 }
```

4.4.3 Data Packets

Data packets are parcels of data that Nidra receives from external applications (e.g., data streams dispatching module) or send to other application (e.g., by sharing the records on the device). This section presents the format of the records and corresponding samples that are sent across applications (e.g., the data is encoded into a file, and the user can select preferred media, such as e-mail). Also, this section presents the data packets forwarded by the data streams dispatching module upon data acquisition from the Flow sensor.

4.4.3.1 Sharing

In Section 4.3.2, a proposal for the structure of exporting and importing data is discussed. Two of the components (`Parse Formatted Content` and `Format File`) use JSON to either encode or decode the data. Listing 4.1 illustrates the content of the encoded data from the application to gain a broader understanding of how the data format looks. To summarize, the JSON string contains an object of record that encompasses the metadata of the recording session, as well as the state of biometrical data of the user. Besides the record object, there is an array of sample objects, where each sample object contains a single sensor reading from the Flow sensor.

```
{
    "record":{
        "id": 1,
        "name": "Record 1",
        "rating": 2.5,
        "description": "",
        "nrSamples": 6107,
        "monitorTime": 5963088,
        "createdAt": "Apr 4, 2019 9:30:56 PM",
        "updatedAt": "Apr 4, 2019 9:30:56 PM",
        "user": {
            "age": 50,
            "createdAt": "---",
            "gender": "Male",
            "height": 180,
            "name": "Ola Nordmann",
            "weight": 60
        }
    },
    "samples": [
        {
            "explicitTS":"Apr 4, 2019 5:51:26 PM",
            "implicitTS":"Apr 4, 2019 7:51:26 PM",
            "recordId":1,
            "sample":"Time=0ms, deltaT=100, data
                =1906,1891,1884,1881,1876,1718,1690"
        },
        ...
    ]
}
```

Listing 4.1: A JSON string object that contains the record which has metadata from the recording session (including the biometrical data of the user), and a list of samples (where each samples contains a single sensor reading).

4.4.3.2 Single Sensor Reading

Listing 4.2 presents a single sensor reading from the Flow sensor sent from the data streams dispatching module. The `id` is assigned by the data streams dispatching module to identify the subscriber application with the corresponding sensor source. Moreover, the `time` is assigned by the internal timer of the Flow sensor. Finally, the `value` is the data sent from the Flow sensor, which contains time, delta, and data. However, we are most interested in extracting the data values from the reading.

```
1 {
2     "id":"1-0",
3     "value":"Time=2100ms, deltaT=100, data
          =1869,1873,1883,1864,1871,1870,1870",
4     "time":"2019-07-29T18:20:58.997+0000"
5 }
```

Listing 4.2: Contains a single sample received from the Flow sensor.

Chapter 5

Implementation

This chapter starts with a brief overview of application components for the three individual applications (i.e., Nidra, data streams dispatching module, and Flow sensor wrapper), alongside a description of the applications. Moreover, the implementation of the separate concerns identified in the previous chapter as an Android application called *Nidra*.

5.1 Application Components

In this thesis, we operate with three individual applications: Nidra, data streams dispatching module, and the sensor wrapper for the Flow sensor. Figure 5.1 illustrates the Android components (i.e., activity, service, provider, and broadcast receiver) for each application, which run in a separate process on the user's mobile device. The subsequent sections present a brief overview of the applications structure and components.

5.1.1 Flow Sensor Wrapper

As part of the thesis, the goal is to integrate the support for the Flow sensor. We developed a sensor wrapper that connects with the sensor source by using the BlueTooth LE protocol. To create a sensor wrapper, we followed the instructions provided by Gjøby [25] in order to create a new driver application that connects with the data streams dispatching module (DSDM). Below, a brief overview of the main components of the driver is discussed.

WrapperService: Is instantiated by the DSDM during the sensor discovery phase (described in Section 5.1.2). This component is responsible for handling starting and stopping of the data acquisition (event sent as broadcasts from DSDM), as well as establish an IPC connection using a binder with the DSDM application.

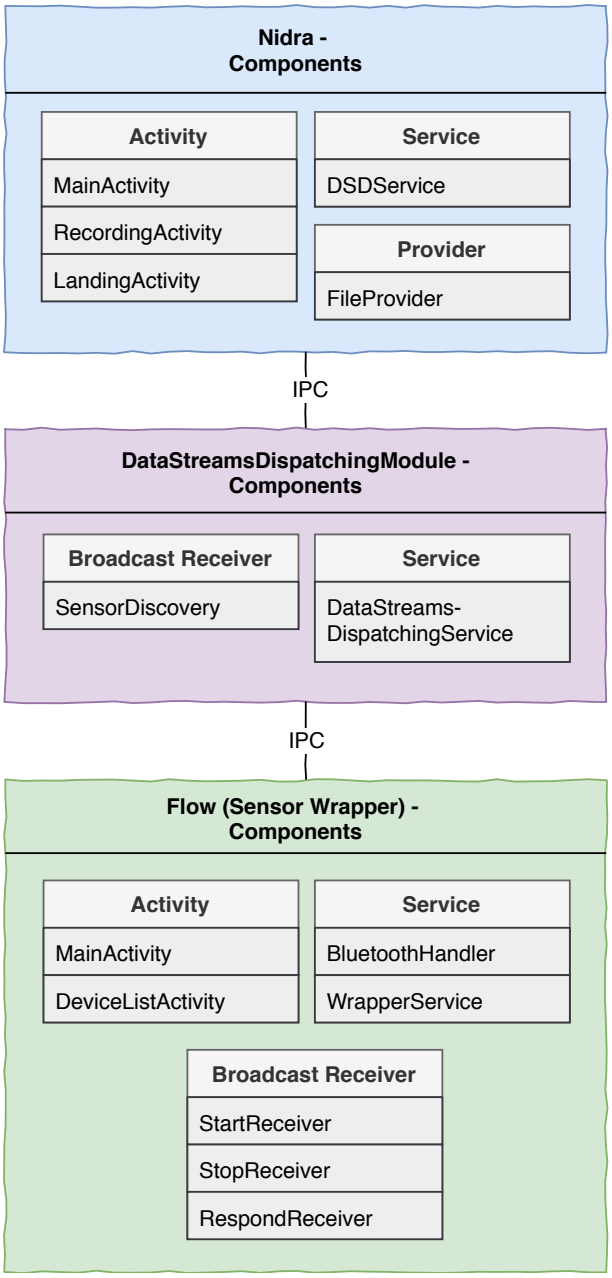

Figure 5.1: Applications components for the three individual Android applications in the project and IPC connection between them.

CommunicationHandler: Upon IPC connection with the DSDM and a request to start data collection, a separate thread of this component is created for interacting with the component that is responsible for integration of BlueTooth LE API's that communicate with the Flow sensor (described in Section 5.1.1.1). However, this component is mainly responsible for forwarding the data acquired from the sensor to the `DataHandler` component.

DataHandler: Preprocesses the collected data from the Flow source into a data packet before forwarding the packet to the DSDM application. Part of this process is to construct the data packet correctly. The data packet is formatted as JSON string and contains the id (Flow) of the sensor wrapper, the current date and time, and the data points from the sensor source. The data packet is then sent, using the established binder connection, to the DSDM application.

Besides the components which manage the connectivity, collection, and disconnection of the Flow sensor with the DSDM, two activities are responsible for providing an interface to select the Flow sensor and display the state of the Flow sensor on the user's screen:

MainActivity: Presents the state and information of the selected Flow sensor on the user's screen. Currently, it presents the connectivity state (connected or disconnected), the battery level of the sensor, the MAC address and the firmware level, as well as the option to remove or connect to another sensor source.

DeviceListActivity: Available devices or sensors that are in range for BlueTooth connectivity with the mobile device is presented to the user. The user has to select the correct Flow sensor that will be used for data acquisition. As a feature, the Flow sensor has a distinguishable icon to make it easier to select the correct sensor source amongst other devices or sensors that is in range of the user's mobile device.

The Flow sensor wrapper stores the name and the MAC address of the selected Flow sensor in a `SharedPreference`. As such, the user has to configure the sensor wrapper once, and the information remains persistent in the application.

The preceding components are a part of the driver to connect with the data streams dispatching module. However, the communication with the sensor source is not a part of these components. The communication with the Flow sensor occurs over BlueTooth LE (BLE) protocol which is designed to provide lower power consumption on data transmission, and sensors that utilize BLE is designed to last for a more extended period. In order to connect with BLE sensors, we can use the API's provided by Android. The implementation of the API's is introduced as a new component (*BluetoothHandler*), which solely communicates with the Flow sensor over BLE. Below, a brief overview of how to establish a connection and interpret the collected data from the Flow sensor through

BLE in Android is described. However, a more detailed description of implementation can be found in Appendix D.

5.1.1.1 Communicating with the Flow Sensor

In order to communicate with the Flow sensor, we have to use the BlueTooth Low Energy (BLE) protocol. To begin with, the user has to select the desired Flow sensor to use for collecting the data. As such, when the command of starting the collection is passed to the `StartReceiver` broadcast receiver which is handled by `WrapperService`, a separate thread of `CommunicationHandler` is created. This thread, start the service of `BluetoothHandler` and initializes the connection to the selected Flow sensor based on the MAC address of the sensor.

The `BluetoothHandler` is the component we introduce, which manages the connection to the Flow sensor, discovers services provided by the sensor, and manages the decoding of the data received from the sensor. This component acts as a GATT client which connects with a GATT server. The GATT server in our case is the Flow sensor, which provides a service that encompasses several characterstics that contains values and descriptors. In BlueTooth, the objects (e.g., services and characteristics) are identified by a universally unique identifier (UUID)[1], and there is a collection of assigned numbers to standard objects [8]. The UUID for GATT attributes for BLE accordingly to BlueTooth is structured as following *PREFIX-0000-1000-8000-00805f9b34fb*, where the prefix is the assigned number that categorizes an individual object. The most interesting characteristics to us, are the manufacturer name (prefix: 0x2A29), firmware revision (prefix: 0x2A26), battery level (prefix:0x2A19) and flow (breathing) measures (prefix: 0xFFB3). The latter characteristic prefix is not a part of the standard; however, the manufacturer defined prefix. Also, it is of most interest to us as it contains the values for the breathing data.

To receive flow (breathing) data and the battery level from the sensor source, we have to enable it by notifying the GATT server. This is performed by specifying the service and the underlying characteristics we want the values from. For example, to enable flow (breathing) data, we specify the service (prefix: OxFFB0) and characteristic (prefix: 0xFFB3), and send it with the API provided by Android. As such, we enable the Flow sensor to collect breathing data.

The Flow sensor gatherers data at a frequency of 10 Hz; however, the data from the sensor is sent to the connected devices on approximately 1.5 Hz. Which means each data received from the sensor contains 5-7 data points with a timestamp of acquisition. We proceed to smooth out the data points by averaging the values, which statistically is to filter out misfits of values and to find an estimate of value on a given time. The data is sent to the `CommunicationHandler` which further sends it to the `DataHandler`. The

[1] A standardized 128-bit format for string ID to uniquely identify information

`DataHandler` creates a data packet with the data as a JSON string, and sends the data packet on the binder (created in the `WrapperService`) between the sensor wrapper and DSDM on the method `PutJson()` (described in Section 5.1.4).

When the command of stopping the collection is passed to `StopReceiving` broadcast receiver, the `CommunicationHandler` thread is interrupted. The interruption closes and unbinds the connection with the `BluetoothHandler`. Within the `BluetoothHandler` the connectivity with the GATT server (sensor) is disconnected and closed. Finally, the screen presented to the user shows that the sensor has disconnected.

To summarize, the `BluetoothHandler` connects with the selected Flow sensor with the BlueTooth LE protocol by using the API's provided by Android. By specifying the appropriate service and characteristic we can obtain breathing data from the sensor. The sensor collects data at a frequency of 10 Hz; however, sends a small burst of data with 5-7 data points for a timestamp at a frequency of 1.5 Hz. Moreover, we procced to smooth out the data points by calculating the average value. The data is then sent to the `CommunicationHandler` which further sends it to the `DataHandler`. The `DataHandler` creates a data packet of value including metadata, and forwards the data packet to the data streams dispatching module.

5.1.2 Data Streams Dispatching Module

The data streams dispatching module developed by Bugajski [12] provides an interface for application (subscribing application) instances to subscribe to data packets from supported and available sensor sources (publishers). The modularity this application provides towards managing and supporting various sensor source allows for faster development time. Also, it provides a common interface for communication with sensor sources[2] that have different Link Layer technology (e.g., BlueTooth) and low-level communcation protocols (e.g., vensors that provides sensor spesific SDK).

This application discovers available sensor wrappers installed on the user's mobile device, establishes connection with the sensor source on requests, and forwards data packets from the sensors to the subscribed applications (all in the `DataStreamDispatchingService` component). Below, we briefly introduce the components and their mechanism in the application.

Sensor Discovery The initial design of discovery for installed sensor wrappers was performed by sending a broadcast with an action of *HELLO* as an intent. All sensor wrappers are designed to respond to this action with their package name as *id* and the name of the sensor wrap-

[2]sensor sources must have a sensor wrapper (e.g., Flow sensor wrapper) in order to work with the data streams dispatching module.

per as *name*. This application is then aware of which sensor wrappers that are available on the mobile device.

However, during the development of this thesis, Android had limited the use of implicit broadcasts on newer Android versions [9]. Implicit broadcasts are those broadcast that does not target a specific application; however, sends out an action with a message and those application that filters and listens for the actions can handle this message accordingly. To overcome this, a re-design of the sensor discovery was made. Instead of DSDM ever so often sends out a HELLO broadcast, the sensor wrapper sends out an explicit broadcast directed to the DSDM to make it aware of its existence. The broadcast is sent to the *SensorDiscovery* directed explicitly to DSDM broadcast receivers, encapsulated with the name and the package name of the sensor wrapper. The DSDM stores the sensor wrapper information in a `SharedPreference`, which will be used locate the sensor wrapper based on the request from the subscribing applications.

DataStreamDispatchingService Encompasses most of the functionality of the application. This component acts as a data distributor between applications that desire to connect with sensor sources, and sensor sources forward their data packets to this service. Also, this component ensures to duplicate and make identical data packets to all subscribed applications. The actions that are exposed to the sensor wrapper and the subscribed applications are managed within this component by providing an interface with a binder for IPC communication (see Section 5.1.4).

To summarize, this application acts as a data distributor for supported sensor sources. Subscribing applications can select a desired sensor source (e.g., Flow) for data acquisition. The data received from the sensor source are obtained in the sensor wrapper and sent to this application as data packets. This application duplicate the data packets to those applications that subscribe to the sensor source and send it to the them accordingly.

5.1.3 Nidra

In this thesis, we focus on creating an interface that can record, share, and analyze breathing data of a patient over an extended period using the Flow sensor. In order to provide an interface, we developed an Android application called Nidra. This section describes the components that Nidra constitute of, while in Section 5.2 we discuss the implementation of Nidra. Below, we describe the main components of the application:

MainActivity: Encompasses most of the functionality (expect recording) in the application, by using fragments as a modular approach to seperate functionality. The fragments lie on top of this host activity, and a transition amongst the fragments is triggered based on user interactions.

RecordingActivity: This activity manages the recording part of the application by invoking the `RecordingFragment`. However, the fragment handles the connectivity with the data streams dispatching module, handles the data packets from the sensor sources, and assuring for reconnecting with the sensor on human disruptions and sensor disconnections. Also, it manages the interactions that can be performed on the recording screen (e.g., real-time graph).

LandingActivity: When launching the application for the first time, a screen with an introduction to the application is shown to the user. Further, the user is prompted to type in biometrical data (i.e., name, gender, age, height, weight), which will be used to enrich the recording of the patient.

Moreover, Nidra leverages the functionality that the data streams dispatching module provides. To use the functionality, the recording activity connects with the data streams dispatching module, and the reference that will be used to receive data packets are sent to the service of *DSDService*. This service implements the interface provided by the data streams dispatching module, and the data obtained within this service are directly sent to the `RecordingActivity` for processing.

Also, it uses the `FileProvider` to securily share files with other applications. That part is used during exporting of records, and the content of the component is found Section 5.3.2.

To summarize, the components discussed in this section constitute an application that enables recording, sharing, and analysis of breathing data obtained from the Flow sensor. Further, we elaborate on the actions and functionality of Nidra in Section 5.2.

5.1.4 Inter-Process Communication

In order to communicate with between the applications, such as remote procedure calls (RPC) to application components that run remotely, we can use the IPC mechanisms. In Android there are two viable mechanisms to enable IPC: (1) `Binder` enables a process to invoke functions in another process remotely; and (2) `Intent` a message passing interface allowing applications to send messages to each other. In this section we describe how these mechanisms are used in Nidra.

The Intent mechanism is mainly used for sharing bundles of primitive data types (e.g., strings or floats) across activities and applications, as long as the reference (e.g., package name) is valid. Another possibility of using intent is during broadcasts within the application or to other application. In Nidra, `Intent`'s are used to start activites, share data between components with local broadcasts, as well as when launching a module (discussed in Section 5.2).

To implement the binder mechanism, we can use design of the data streams dispatching modules which use Android Interface Definition Language (AIDL). In order to communicate with processes, the data objects have to be decomposed into primitives that the operating system can understand. AIDL provides this mechanism by providing a programming interface that both the client and the service agree upon. The AIDL interface is defined in an `.AIDL` file, and located in the `src/` directory of the hosting service application (DSDM), likewise, for other applications that bind to the hosting service (Nidra and sensor wrappers). It is essential to have identical `.AIDL` files across the applications, otherwise the system will not recognize it as the same interface. In Listing 5.1, the interface is based on the functionality of the hosting service application provides (DSDM).

```
1  // MainServiceConnection.aidl
2  package com.sensordroid;
3
4  interface MainServiceConnection {
5      void putJson(in String json);
6      int Subscribe(String capabilityId, int frequency, String
           componentPackageName, String componentClassName);
7      int Unsubscribe(String capabilityId, String
           componentClassName);
8      String Publish(String capabilityId, String type, String
           metric, String description);
9      void Unpublish(String capabilityId, String key);
10     List<String> getPublishers();
11 }
```

Listing 5.1: An interface provided by the host service (i.e., DSDM) that provides functionality other applications can use (e.g., Nidra and sensor wrappers)

In Nidra, some of the functionality is utilized to enable recording. More specifically, `getPubishers()` method is used to get all of the sensors publishers (e.g., Flow sensor), the `Subscribe()` and `Unsubscribe()` is used in order to subscribe and unsubscribe to a specific sensor publisher, and the data that is forwarded from by the sensor publisher sent to the `putJson()` method.

As for the Flow sensor wrapper, uses the same interface to communicate with the data streams dispatching module. However, the only method call is towards `putJson()` when forwarding its data packets. The data streams dispatching module is aware of which type of connectivity (e.g., publisher or subscriber) that forwards the data, therefore, the call by the sensor wrapper is processed as data packets that will be sent to the subscribing applications.

5.2 Implementation of Concerns

In the design chapter of this thesis, we conceptualized the tasks by decomposing them into separate concerns and discussing design choices for implementation. This section realizes the discussion by implementing the concerns in Android and developing the application Nidra. For each concern, we illustrate the components and objects that interact with each other, step-by-step; the legend for the figures are shown in Figure 5.2.

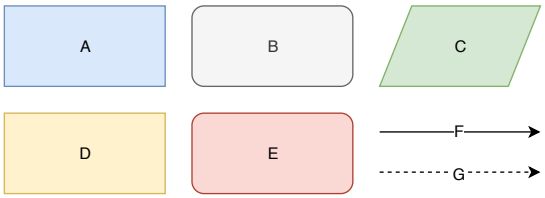

Figure 5.2: Legend for the figures in implementation of concerns: (A) application components with integration of our logic; (B) objects that contains specifics of our logic; (C) an interface for callbacks or listeners; (D) Android-specific objects and components; (E) other installed applications; (F) step direction; and (G) reference or data flow direction.

5.2.1 Recording

The design choices for this concern is described in Section 4.3.1. To summarize, recording is the process of collecting and storing data received from sensors over an extended period. To enable a recording, we need to establish a connection with the available sensors and store the samples retrieved by the sensors on the device. In this thesis, we focus on collecting breathing data from the Flow sensor. Moreover, we use the data streams dispatching module (hereafter: DSDM), which manages sensor discovery and sensor establishment to supported sensor sources. The DSDM facilitates an interface for data acquisition, and the communication between the DSDM and Nidra occurs over IPC using binder's. During recording, we check for connectivity with the sensors to ensure the sensors are collecting data at an appropriate rate. At the end of the recording, we store metadata related to the recording and finalize the recording process.

The functionality of recording are separated into three actions: (A) start recording; (B) stop recording; and (C) display recording statistics. The following sections review the steps that enable these actions.

5.2.1.1 Action A: Start Recording

In Figure 5.3, an illustration of the component interactions are shown. Action A is to start a recording by connecting and starting data aqusition

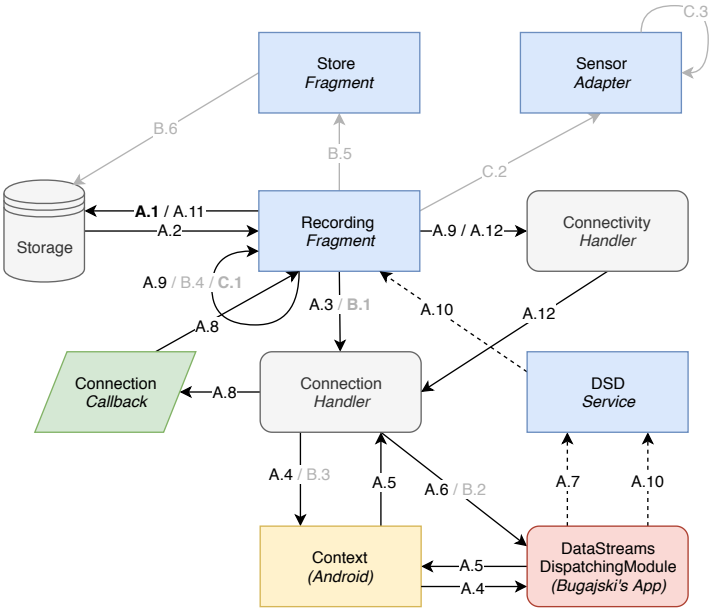

Figure 5.3: Implementation of the recording action (A): start recording.

with the use of DSDM, and to ensure that the sensor source is gathering data at an appropriate rate. The steps and interactions for this action are:

A.1 The recording process starts by creating a new record entity that is inserted into the storage (see Section 5.2.5). An empty record has to be inserted into the database in order to associate new samples with the record (based on the record's id).

A.2 Once the record is inserted into the storage, a unique identification (id) is returned.

A.3 `ConnectionHandler` is invoked in order to manage the establishment, connection, and disconnection of the IPC between Nidra and DSDM service. The code for establishing the connection is described in the following listing (MainServiceConnection is the AIDL file as discussed in Section 5.1.4):

```
1    Intent intent = new Intent(MainServiceConnection.class.
         getName());
2    intent.setAction("com.sensordroid.ADD_DRIVER");
3    intent.setPackage("com.sensordroid");
4    context.bindService(intent, serviceCon, Service.
         BIND_AUTO_CREATE);
```

A.4 If the service is offline when binding, the flag `Service.BIND_AUTO_CREATE` will ensure for starting the service. `BindService` allows components to send requests, receive responses, and perform inter-process communication (IPC) based on the interface provided by the host service

(DSDM).

A.5 Once the service is bound, we can proceed to communicate with the DSDM service.

A.6 The `ConnectionHandler` proceeds to initialize the connection with the sensor through the DSDM. A request to the DSDM for available publishers with `getPublishers()` is made, to retrieve all available sensor publishers connected to the DSDM. Occasionally, the DSDM uses extended time to discover all of the active sensors connected to the device; therefore, we have an interval that checks whether DSDM has any available sensors connected.

A.7 Moving on, a request to the DSDM to `Subscribe` to a sensor is made. We specify that we want the Flow sensor in the `Subscribe` method, in addition, a reference to the package name (Nidra) and a service object (`DSDService`). The service object is where all of the data packets from the subscribed Flow sensor is received (on the `putJson()` method).

A.8 Also, a callback to `RecordingFragment` with information of the sensor source (i.e., Flow) that we subscribe to is made, in order to display the information on the user's screen.

A.9 The recording has now started, and a timer to measure the time spent on the recording is started. The `ConnectivityHandler` is also initialized, which actively checks that the samples arrive within a specified time frame (as discussed in the design, a frame of 10 seconds that increases throughout the recording). The `ConnectivityHandler` is implemented with a `Handler` with a `PostDelay` that counts down to zero. Upon a sample arrival, the timer is reset.

A.10 Periodically, the DSDM receives samples from the Flow sensor. DSDM forwards the sample from the sensor to the service object (`DSDService`) on the `putJson()` method. The DSDService uses a `LocalBroadcastManger` to send the data packet to the `RecordingFragment`. This process goes on infinitely unless the recording or the sensor has been stopped.

A.11 `RecordingFragment` listens for the events on the local broadcast receiver. Upon an event, the data that is received from the sensor is processed. In other words, we extract the samples from the data packet, and insert it as a new a new sample entity with the current record's id as an association.

A.12 Recalling the functionality from step A.9; if the event for the `PostDelay` is triggered, it is equivalent to a sample not being acquired from the sensor. Therefore, we try to reconnect with the subscribed sensor by disconnecting with the DSDM (which will close the connection with the sensor source), followed up by a reconnection with the DSDM and subscribing to the same sensor source. Most of the times, the process of reconnecting with the sensor one time

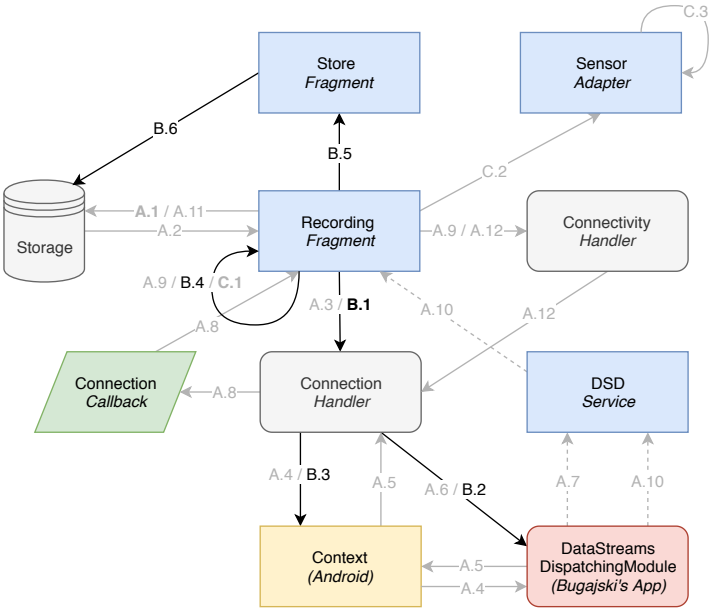

Figure 5.4: Implementation of the recording action (B): stop recording.

solves the issue; however, some times the sensor might require to be reconnected with several times in order to work.

5.2.1.2 Action B: Stop Recording

In Figure 5.4, an illustration of the component interactions are shown. Action B is based on user input to stop the recording process. To the user, the recording has terminated, and the user is presented with a screen to provide extra information regarding the recording (e.g., title, description, and rating). For the application, it has to unsubscribe from the connected sensor sources (i.e., Flow), and disconnect the connectivity with the DSDM. The steps and interactions for this action are:

B.1 The user decides when to stop a recording with a press of a button. The event to stop the recording is sent to the `ConnectionHandler`.

B.2 A call to the `Unsubscribe()` method that contains the service object (i.e., DSDService) and the identification of the sensor source (e.g., Flow) is sent to DSDM. The DSDM has to ensure unsubscribing the sensor from a specific application and disconnect the IPC between the application. If there are no subscribing applications to the specific sensor source, the DSDM will signal the sensor to stop sampling and disconnect with the sensor.

B.3 The IPC connection between Nidra and DSDM is discontinued by unbinding the service.

B.4 The estimated time of recording is calculated, and a transition from `RecordingFragment` to `StoreFragment` is made to finalize the

recording with extra information (e.g., title, description, and rating).

B.5 The `StoreFragment` uses the record identification retrieved on recording (A.1) in order to update the record with statistics and user-defined metadata. The statistics are the monitoring time, number of samples during recording, and retrieving the current state user biometrical data. The user-defined metadata are the title of the recording, a description enabling the user to add a note to the recording, and a rating between 1–5 (to give a rating on how the recording felt).

B.6 The modified record is updated in the database, and the user is transitioned to the `MainActivity`.

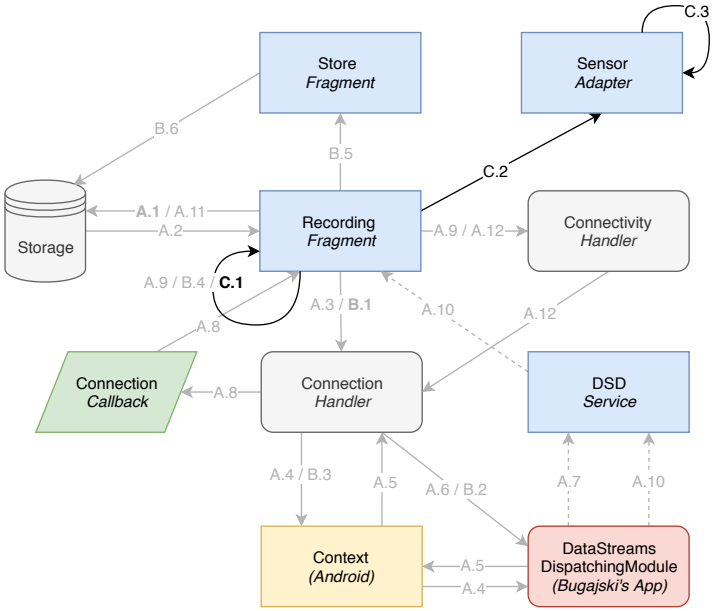

Figure 5.5: Implementation of the recording action (C): display recording statistics.

5.2.1.3 Action C: Display Recording Statistics

Action C is to display statistics during a recording as a separat interface overlay. More specifically, the user can see the available connected sensors and a graph of the breathing data in real-time. In Figure 5.5, an illustration of the component interactions are shown, and the steps and interaction for this action are:

C.1 The data is graphically represented as an intractable time-series graph. By using the GraphView library [5], we can in similarities to the implementation of the analytics concern (see Section 5.2.4), implement a graph to illustrate the breathing data to the user in real-time. As such, the user is presented with an interactable graph that continously updates with the samples acquired from the sensor.

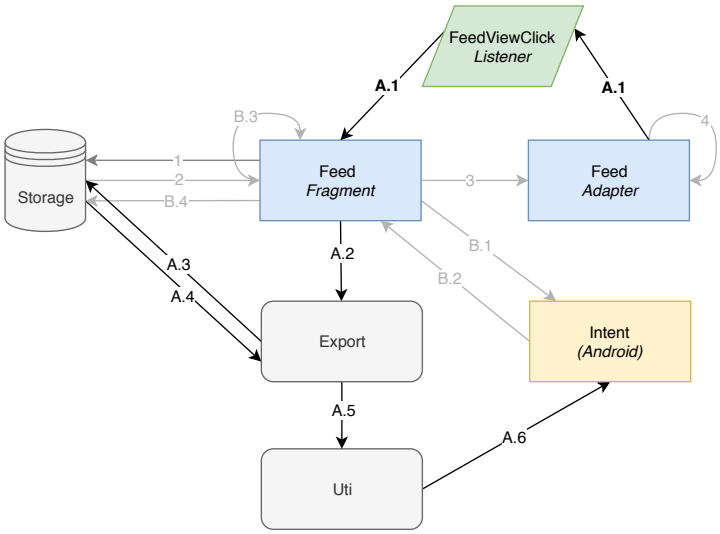

Figure 5.6: Implementation of the sharing action (A): exporting one or all records.

C.2 In addition to the time-series graph, we have a list of publishers (e.g., Flow) that we acquired in the beginning of the recording. As such, a list of publishers is sent to `SensorAdapter`.

C.3 The `SensorAdapter` populates a view with the connected sensor to the user, in our case the Flow sensor.

5.2.2 Sharing

The design choices for this concern is described in Section 4.3.2. To summarize, sharing enables users to transmit records across application over a media (e.g., e-mail). The functionality of sharing is separated into two concerns, namely exporting and importing. The exporting consists of packing the records the user has selected for transmittal to another mobile device, while importing consists of locating the file on the user's device and parsing the data and storing in the database.

Before a user can proceed with these actions, the records from the database have to be presented. The `FeedFragment` contains a `RecyclerView` which populates the records into inside the `FeedAdapter` (steps: 1-4). The adapter contains all the interactions and the event handling (e.g., button event listener for exporting) for a single record.

The functionality of sharing are separated into two actions: (A) exporting one or all records; and (B) import a record from the device. The following sections review the steps that enable these actions:

5.2.2.1 Action A: Exporting one or all Records

In Figure 5.6, an illustration of the steps to export one single recording is shown. However, the `Feed Fragment` has an option to export all record; therefore, by disregarding the first step (A.1), the same structure applies to export all records. In essence, exporting consists of bundling the records and corresponding samples into a formatted file, and prompting the user with options to select a media (e.g., mail) for transmission. The steps are narrowed down to:

A.1 Upon an event for exporting a selected record in `FeedAdapter`, the record information is sent to the `FeedFragment` through the callback reference (`onRecordAnalyticsClick`) between these components. The record information will be used to determine the corresponding samples for the record.

A.2 The `FeedFragment` sends the record information to the `export` method inside of the `Export` object, which is responsible for enabling the export.

A.3 An operation to retrieve all samples related to the record (see Section 5.2.5) is done.

A.4 The `export` method retrieves all of the samples related to the record. Next, the record and the samples are encoded into an exportable JSON format (see Section 4.4.3). In order to share files between applications, the content has to be stored on the device. Thus, the encoded data is written into a file on the device, with a filename of `record_(current_date).json`, and the next step uses the reference to the file location.

A.5 The encoded file URI is retrieved with the use of `FileProvider` (facilitates secure sharing of files across applications). The code snippet for this step is shown in the following listing:

```
1  static void shareFileIntent(Activity a, File file) {
2      Uri fileUri = FileProvider.getUriForFile(a.
           getApplicationContext(), a.getApplicationContext().
           getPackageName() + ".provider", file);
3
4      Intent iShareFile = new Intent(Intent.ACTION_SEND);
5      iShareFile.setType("text/*");
6      iShareFile.putExtra(
7          Intent.EXTRA_SUBJECT, "Share Records");
8      iShareFile.putExtra(Intent.EXTRA_STREAM, fileUri);
9      ...
10     a.startActivity(
11         Intent.createChooser(iShareFile, "Share Via"));
12 }
```

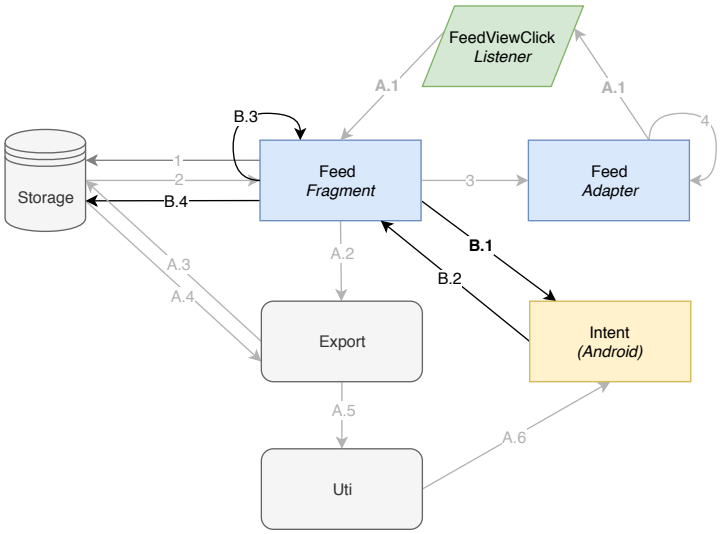

Figure 5.7: Implementation of the sharing action (B): import a record from the device.

A.6 The user is displayed with a popup interface with several options to share the file over a media (e.g., e-mail). An illustration of the layout is found in Section 5.2.6.

5.2.2.2 Action B: Import a Record from the Device

In Figure 5.7, an illustration of importing a record from the device is shown. Importing consists of locating the formated file (the user has to obtain the file and store it on the device on beforehand), parsing the content in the file, and storing the data respective to the user's database. The steps are narrowed down to:

B.1 The user requests to view the import interface. The interface is provided by Android and allows the user to select files on the device. The code snippet for this step is shown in the following listing:

```
1 private void importRecords() {
2     Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
3     intent.setType("*/*");
4     startActivityForResult(intent, 1);
5 }
```

B.2 Once the user has selected the desired file, the method `onActivityResult` inside of `FeedFragment` is called, and location of the selected file can be obtained.

B.3 The file location is an obscured path to the file on the device; thus, parsing the path with the use of `Cursor` method (Android library)

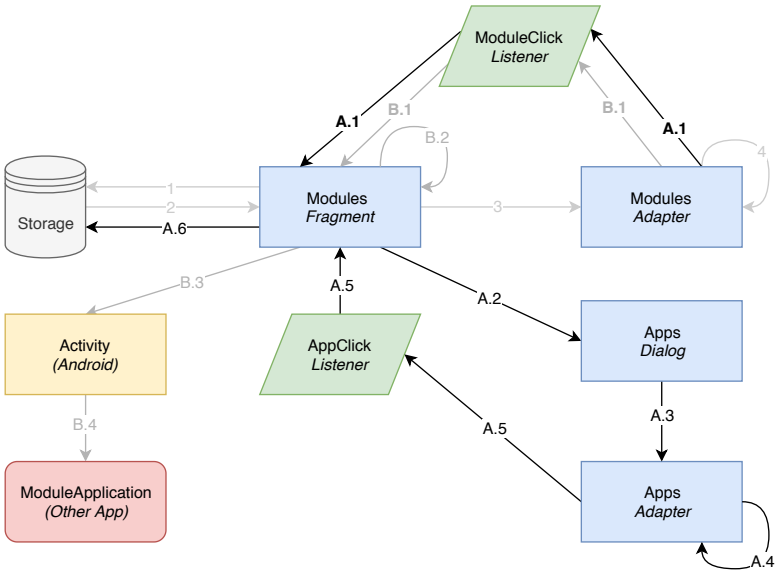

Figure 5.8: Implementation of the module action (A): add a module.

has to be done. After the absolute path is found, the data is decoded accordingly to the data format.

B.4 The record information and the samples are extracted from the decoded data. The information is then put into a new record entity alongside new samples entities, and are inserted into the user's database.

5.2.3 Modules

The design choices for this concern is described in Section 4.3.3. To summarize, modules are standalone application that provides data enrichment or extended functionality to Nidra. For example, a module can use the records with samples provided by Nidra in order to feed a machine learning algorithm that predicts sleep apnea. Moreover, to add and launch a module from Nidra we need the modules package name. The package name and the name of the module-application is easily obtained within Android.

The functionality of modules are separated into two actions: (A) add a module; and (B) launch a module. The following sections review the steps that enable these actions.

5.2.3.1 Action A: Add a Module

In order to add a new module, the user has to install the module-application on the device beforehand. By listing through the installed application on the device, the user can select the desired module to be

added in Nidra. In Figure 5.8, an illustration of adding a module is shown, and the steps are narrowed down to:

A.1 Upon an event for adding a new module in `ModulesAdapter`, the `ModulesFragment` is notified through the callback reference (`onNewModuleClick()`) between these components.

A.2 The `ModulesFragment` launches a custom Android dialog, which will list all of the installed application on the device.

A.3 The `AppsAdapter` will fetch all of the application that is not a system package, already an installed module, or the current application (Nidra). Next, the adapter for the dialog will be populated with the eligible applications.

A.4 Once the user has selected the desired module-application, an event to the `ModulesFragment` through the callback reference `onAppItemClick()` between these components are made. The callback contains an Android object of `PackageInfo` (contains the name and package name of the application) for the selected module-application.

A.5 The dialog is dismissed, and the application name and package name are extracted from the `PackageInfo` for the selected module-application.

A.6 Furthermore, the acquired information (i.e., name and package name) is stored in our database as a module entity.

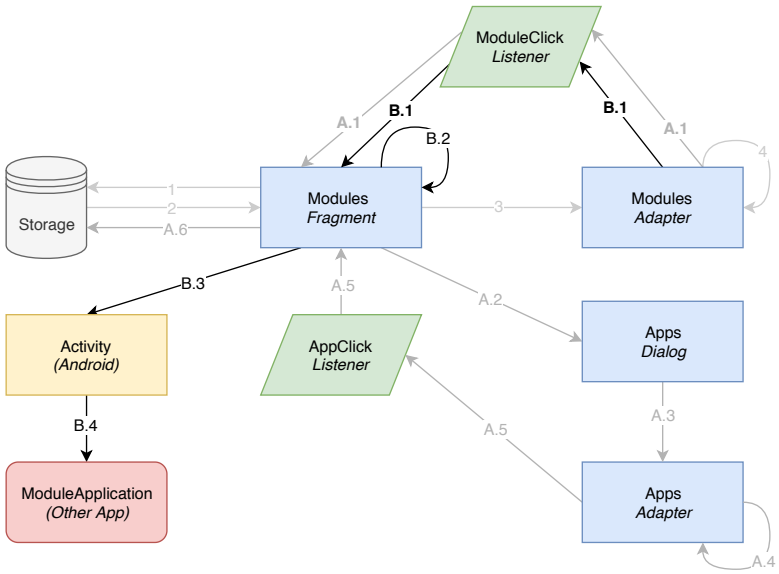

Figure 5.9: Implementation of the module action (B): launch a module.

5.2.3.2 Action B: Launch a Module

A module is launched as a separate application with a separate process, due to Android prohibits launching for other applications inside of an application. All added modules are listed and presented to the user on a separate screen. On the launch of a module, all records (including corresponding samples) in Nidra are encoded into a JSON format and bundled with the launch of the module. In Figure 5.9, an illustration of launching a module, and the steps are narrowed down to:

B.1 Upon an event for launching a module in `ModuleAdapter`, the package name of the module is sent to the `ModulesFragment` through the callback reference (onLaunchModuleClick()) between these components. The package name will be used to launch the module-application.

B.2 All records (including their samples) on the device for the user, are formated into a JSON string and bundled into the launch. The code snippet for this step is shown in the following listing:

```
1  public void onLaunchModuleClick(String packageName) {
2      Intent moduleApplication = context.getPackageManager().
           getLaunchIntentForPackage(packageName);
3
4      if (moduleApplication == null) return;
5
6      String data = formatAllRecordsToJSON();
7
8      Bundle bundle = new Bundle();
9      bundle.putString("data", data);
10
11     moduleApplication.putExtras(bundle);
12
13     startActivity(moduleApplication);
14 }
```

B.3 The activity uses the data provided in the `Intent`, which includes the package name (the name of the module-application to determine the correct application) to launch the application with `startActivity()`.

B.4 The selected module is then launched and presented to the user. The user can at any time press the back button to return to Nidra.

5.2.4 Analytics

The design choices for this concern is described in Section 4.3.4. To summarize, analytics is the part of illustrating and analyzing the records. In

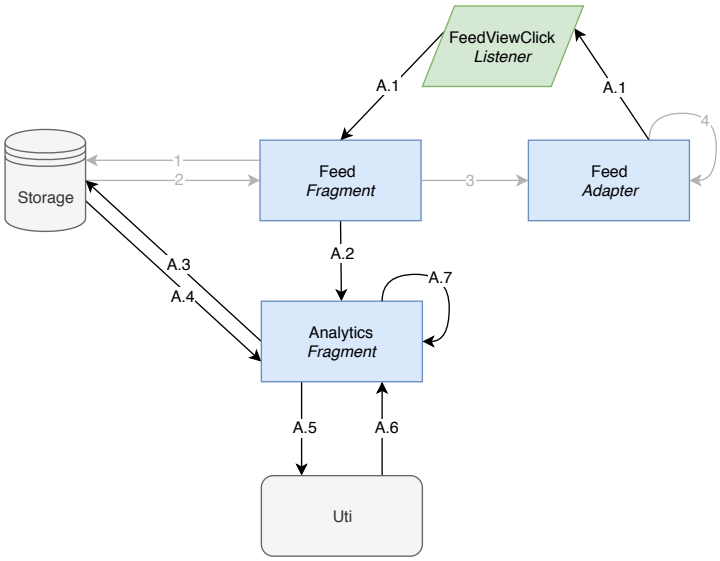

Figure 5.10: Implementation of the analytics action (A): display a graph for a single record.

Nidra, the analytics part of the implementation is limited to a time-series graph for a single record. While there are possibilities of extending the `AnalyticsFragment` with other graphs based on the current structure, the facilitation of modules allows for easier integration of various analytical techniques and methods.

Similar to sharing, the records from the database have to be presented. The `FeedFragment` contains a `RecyclerView` which populates the records into inside the `FeedAdapter` (steps: 1-4). The adapter contains all the interactions and the event handling (i.e., button event listener for analytics) for a single record.

The functionality of analytics is identified by one action: (A) display a graph for a single record to the user. The following sections review the steps that enable this action.

5.2.4.1 Action A: Display a Graph for a Single Record

Nidra provides a simple time-series graph of breathing data obtained during the recording. The graph data is plotted into a GraphView library [5], which enables interactions (e.g., zoom and scrolling) on the data samples. The X-axis is the breathing value based on the Y-axis of time of sampling. In Figure 5.10, an illustration of displaying a graph shown, and the steps are narrowed down to:

A.1 Upon an event for analytics on a selected record in `FeedAdapter`, the record information is sent to the `FeedFragment` through the callback reference (`onRecordAnalyticsClick()`) between these components.

The record information will be used to determine the corresponding samples for the record.

A.2 A new instance of the `AnalyticsFragment` is created, and a transition from the `FeedFragment` to the `AnalyticsFragment` is made. Alongside, the record information is transmitted as a bundle.

A.3 An operation to retrieve all samples related to the record with the use of the `SampleViewModel` is done.

A.4 The `AnalyticsFragment` retrieves all of the samples related to the record. The samples have to be structured according to the graph library to display an interactive time-series graph.

A.5 Each sample has to be extracted from the sample-data according to the sensor data structure (see Section 4.4.3).

A.6 The sample value is returned and inserted into an array over data points used in the graph.

A.7 The user is presented with a graph which is intractable. The Y-axis has the sample value on the given time (in HH:MM:SS) on the X-axis. The graph library enables interactions (e.g., zooming and scrolling) to gain a better understanding of the recording.

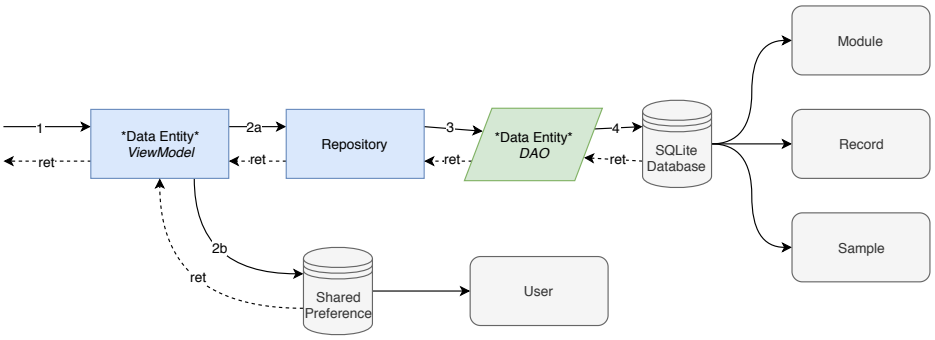

Figure 5.11: Implementation of the storage concern.

5.2.5 Storage

Storage facilities persistent data which remain on the device after application termination. In Nidra, there are four individual data entities (i.e., record, sample, module, and user). In Section 4.3.5, we discussed the design choices of storage possibilities for the individual data entity. To summarize, the entities of record, sample, and module are stored in a SQLite database, while the user entity is stored in a SharedPreference on the device.

The data entities has support for the standard CRUD operations (i.e., create, read, update, and delete); however, some of the entities has extra operations: record entity has an operation to retrieve all of the records on the user's device, sample entity has an operation to retrieve all of the

samples for a given record (based on record's id); the module entity has an operation to retrieve all of the samples on the user's device.

Android Room provides an abstract layer over SQLite to enable easy database access [45]. In Figure 5.11, the flow for accessing and retrieving the data from the database based on the Android Room architecture is shown and are described as:

1 Each data entity has a `ViewModel` where all of the CRUD operation goes through. A view model is designed to store and manage UI-related data in a conscious way, that allows data to be persistent through configuration changes (e.g., screen rotations).

2a The predefined operations point to the repository. Repository modules handle data operations and provide an API which makes data access easy. A repository is a mediator between different data sources (e.g., database, web services, and cache). In Nidra, the only data source is the database, but repository facilities future data sources.

2b The storage of the user is not in the database; however, in a `SharedPreference` on the device. Shared preference points to a file containing key-value pairs and provides the standard CRUD operations. The location of the user's shared preference is `no.uio.cesar.user_storage`.

3 Each data entity (disregarding user) has a data access object (DAO), where the SQL operations to the database are defined.

4 Based on the operation, the data is either insterted, updated, deleted or retrieved from the SQLite database located on the user's mobile device.

Conclusively, all of the preceding concerns (i.e., recording, sharing, modules, and analytics) access the storage as described in this section. As such, future operations can easily be integrated into described components, which enforce modularity and extensibility to the application.

5.2.6 Presentation

The interface is developed based on the design descisions made in Section 4.3.6, as well as creating a user-interface that is simple and efficent for a user to interact with. We try to limit the actions the user can take on a screen, to make the application simpler to understand and comprehend. The following sections present the user-interface (UI) based on the functionality of recording, sharing, module and analytics.

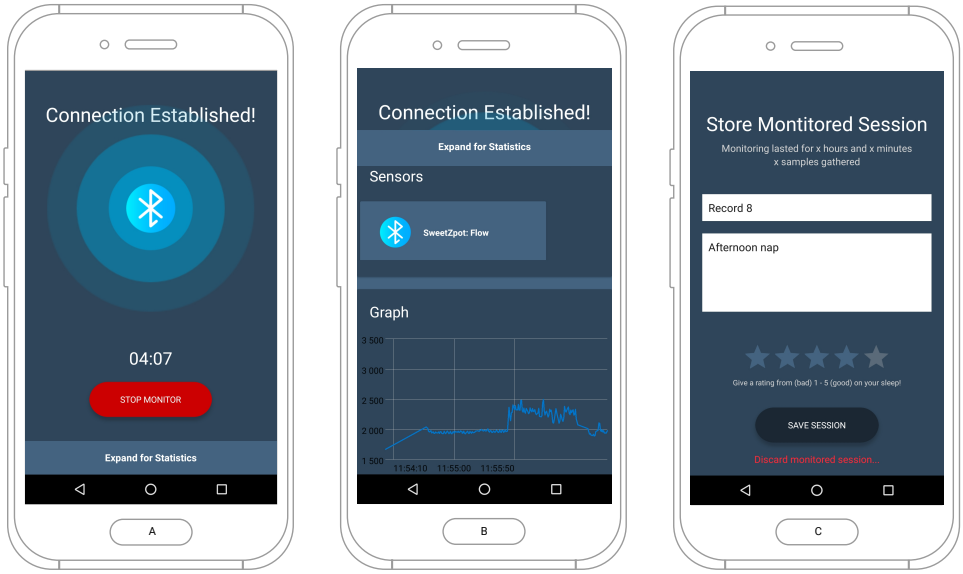

Figure 5.12: The recording screen displayed to the user: (A) during a recording, (B) statistics interface, and (C) stopping the recording.

5.2.6.1 UI: Recording

Figure 5.12 presents the screen for (A) the screen displayed during recording, (B) the interface for statistics, and (C) the screen displayed at the end of the recording.

A The screen displays the elapsed time of recording right below the center of the screen. The button color has a different nuance from the background color to make more distinguishable to the user. On button click, an alert specifying whether the recording should be stopped or not is presented to the user in order to prevent undesired stopping of recording due to misclicks.

   Moreover, the screen has a ripple-effect to indicate the state of the recording to the user. There are two types of ripple-effect colors: a blue ripple-effect indicates for samples acquisition, while a grey ripple-effect indicates that the sensor has disconnected or trying to reconnect. The ripple effect is only active if the screen is turned on in order to preserve battery life.

   As a side note, during disconnects between the sensor and the device, the user provides no extra input to resolve the issue. The attempts to reconnect occurs in the background, and the only state of change is the ripple-effect color.

B The user can expand the interface for viewing statistics with information about the recording. Currently, the interface lists all of the available sensor sources, also a real-time interactable graph of breathing data.

C The finalizing screen allows users to specify the title and description of recording, as well as giving a rating between the values of 1–5 (where one is bad, and five is good). Upon saving the recording, the user is transitioned to the `MainActivity`.

5.2.6.2 UI: Sharing

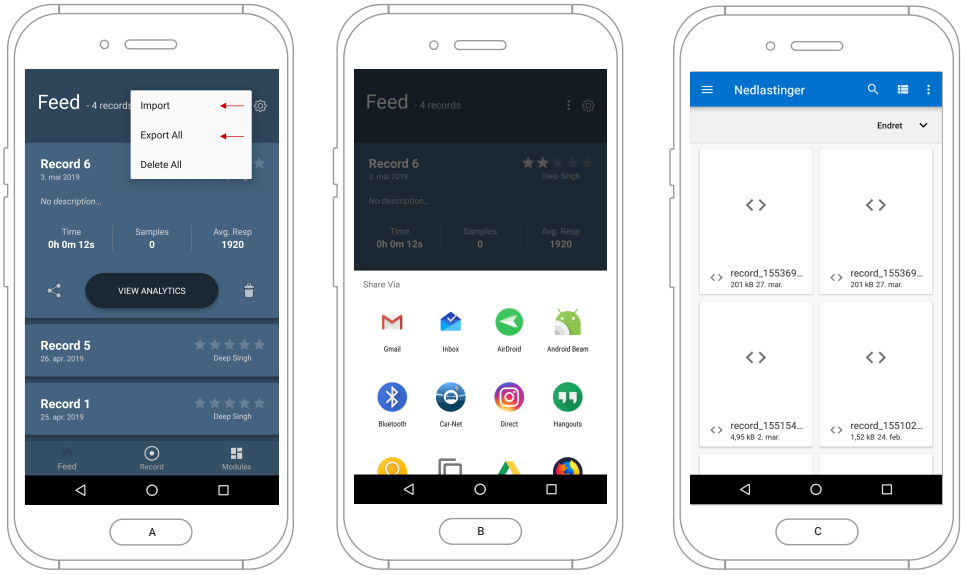

Figure 5.13: The sharing screen displayed to the user: (A) option to import or export, (B) the media selection for exporting, and (C) the file selection for importing.

Figure 5.13 presents the screen for: (A) option to import or export, (B) the media selection for exporting, and (C) the file selection for importing.

A The feed screen is where all of the records are presented to the user as a list. The items in the list are expandable and collapsible to prevent fluttering on the screen, as well as displaying the most important information to the user. The user can press the share icon on a given record, or press on the options menu at the top of the screen to export all or import records.

B By pressing export all (or export on a single record), an overlay with an Android provided sharing screen is presented. The interface lists of all applications that provide a method of sharing data, and to us, the e-mail application is the most interesting method to use for sending records.

C By pressing import, an interface that presents all of the downloaded files on the user's device is shown to the user. The user can press on the desired file, and the file will be parsed and added to the user's collection of records.

5.2.6.3 UI: Modules

Figure 5.14: The module screen displayed to the user: (A) module screen without any modules, (B) list of installed applications on the device, and (C) module screen with modules.

Figure 5.14 shows the screen for (A) module screen without any modules, (B) list of installed applications on the device, and (C) module screen with modules.

A The modules screen without any installed modules; however, a button for adding new modules.

B The user can press the *add new module* button, in order to be presented with a list of all installed applications.

C Once the user has selected a module-application to be added in Nidra, the list of modules are updated and presented to the user. The modules are hereafter added in Nidra and can be launched by clicking on the module. The module can also be removed by long-pressing (holding down for at least 2 seconds) the module-button.

5.2.6.4 UI: Analytics

Figure 5.15 shows the screen for (A) feed screen with a record expanded and (B) the analytics for the record.

A The user can expand the records to view more information and actions. One of the actions is to view the analytics for the record.

B A new screen with an interactable graph, that is populated with the samples associated with the selected record, is presented to the users. The graph shows the respiration (breathing) value (Y-axis) on

given time (X-axis) of sampling. Also, information on the number of samples and elapsed time of recording.

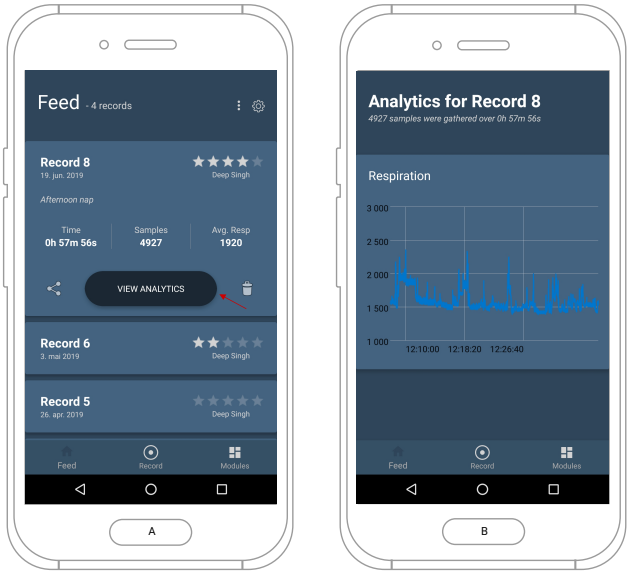

Figure 5.15: The analytics screen displayed to the user: (A) the feed screen; (B) the analytics screen.

5.3 Miscellaneous

5.3.1 Collecting Data Over a Longer Period

In Android, applications which are idle in the background or not visible to the user can be killed in order to reclaim resources for other applications or preserve battery time. However, this mechanism is not viable for collecting data over an extended time, because it can kill our applications during recording. To overcome this, there are several methods to prevent the Android system from killing our applications, which is presented in the subsequent sections.

5.3.1.1 Keep the CPU Alive

The Android system provides a wake lock mechanism to keep the CPU running in order to complete work. As long as we keep the CPU alive, we can collect the data over an extended period. Any applications can utilize wake-locks in their application; albeit, the documentation states that holding onto a wake lock for a longer period, shortens the device's battery time. Therefore, it is crucial to release the lock when the recording has terminated. Also, to use wake-locks the permission has to be added in the application's manifest file (see Section 5.3.2). Nidra utilizes the wake lock

when the recording has started (inside of the `RecordingFragment`) and are seen in following listing:

```
1 powerManager = (PowerManager) mContext.getSystemService(Context
      .POWER_SERVICE);
2 wakeLock = powerManager.newWakeLock(PowerManager.
      PARTIAL_WAKE_LOCK,
3         "CESAR::collection");
4
5 wakeLock.acquire();
```

The lock is released when the activity is destroyed by terminating the recording process.

5.3.1.2 Priority

Process's lifecycle is not directly related with the host application; however, determined by the system which detects parts of applications that are running, how important they are to the user, and how much memory is available in the system. A process can be killed by the system to reclaim memory for other processes to take its place. However, there are specific measures to prolong the services run time. That is, to increase the process importance in the "process-hierarchy". By assigning a process to be a *foreground process*, we can, for most cases, prevent the system from killing a process. In our case, the `DSDService` which receives data packets from the data streams dispatching module. In the following listing, the code snippet for creating a foreground process is presented.

```
1  public void toForeground() {
2      NotificationManager notificationManager =
3          (NotificationManager) this.getSystemService(Context.
               NOTIFICATION_SERVICE);
4      NotificationCompat.Builder builder = null;
5      if (android.os.Build.VERSION.SDK_INT >= android.os.Build.
           VERSION_CODES.O) {
6          int importance = NotificationManager.IMPORTANCE_DEFAULT;
7          NotificationChannel notificationChannel = new
               NotificationChannel("ID", "Name", importance);
8          notificationManager.createNotificationChannel(
               notificationChannel);
9          builder = new NotificationCompat.Builder(this,
               notificationChannel.getId());
10     } else {
11         builder = new NotificationCompat.Builder(this);
12     }
13
```

```
14      builder.setSmallIcon(R.drawable.ic_info_black_24dp);
15      builder.setContentTitle("Nidra");
16      builder.setTicker("Recording");
17      builder.setContentText("Recording data");
18
19      Intent i = new Intent(this, DSDService.class);
20      i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.
            FLAG_ACTIVITY_SINGLE_TOP);
21      PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
22      builder.setContentIntent(pi);
23
24      final Notification note = builder.build();
25
26      startForeground(android.os.Process.myPid(), note);
27  }
```

5.3.2 Android Manifest

The Android Manifest describes the essential information about our application, such as the application components, permissions, and the package name. The application is constituted by application components, and each component contains metadata describing the application component. Below, we describe the some of permissions and a few application components of Nidra and the Flow sensor wrapper.

5.3.2.1 Nidra

The Nidra manifest file constitutes of three activities, one service, and one provider. The latter is used to share a record between applications. Providers enable other applications to access a file or data from Nidra. With the provider, a direct URI link obtained by the provider grants a more secure sharing of data between application. In the listing below, the attribute `authorities` is the name that identifies the data offered by the provider (often distinguished by package name and postfix of "provider"). Also, the meta-data with the resource contains information with the path to the file in the respective application directory.

```
1  <provider
2      android:name="androidx.core.content.FileProvider"
3      android:authorities="${applicationId}.provider"
4      android:grantUriPermissions="true">
5      <meta-data
6          android:name="android.support.FILE_PROVIDER_PATHS"
7          android:resource="@xml/provider_paths" />
8  </provider>
```

The permissions for Nidra are presented in the listing below, which includes the wake lock permissions, and the permissions to store data in external storage and the internal storage. The storage permissions are required in order to use the `SharedPreference` method for storing, as well as the storage of files that will be accessed by other applications (during sharing of records).

```
1 <uses-permission android:name="android.permission.
    WRITE_EXTERNAL_STORAGE" />
2 <uses-permission android:name="android.permission.
    READ_EXTERNAL_STORAGE" />
3 <uses-permission android:name="android.permission.WAKE_LOCK" />
4 <uses-permission android:name="android.permission.
    READ_INTERNAL_STORAGE" />
```

5.3.2.2 Flow Sensor Wrapper

As for the Flow sensor wrapper, the application components structure are based on the driver created by Gjøby [25]. With an expectation of the activities and the BlueTooth service. In the listing below, the permissions of the application are shown. In order to leverage the Bluetooth LE protocol, we need the permissions of *BLUETOOTH*, *BLUETOOTH_ADMIN*, *ACCESS_FINE_LOCATION*. The latter permission is obligatory because it is used to list the available sensor source in the area. Without permission, Android does not present any list of available sensor sources.

```
1 <uses-permission android:name="android.permission.BLUETOOTH"/>
2 <uses-permission android:name="android.permission.
    BLUETOOTH_ADMIN"/>
3 <uses-permission android:name="android.permission.WAKE_LOCK"/>
4 <uses-permission android:name="android.permission.
    ACCESS_FINE_LOCATION"/>
5 [...]
```

Part III

Evaluation and Conclusion

Chapter 6

Evaluation

In this chapter, experiments are conducted in order to evaluate the application based on the system requirements defined in the problem statement:

1. The application must provide an interface for the patient to (i) record physiological signals (e.g., breathing data); (ii) present the results; and (iii) share the results.
2. The application must ensure that upon sensor disconnections, the connectivity is reinstated to minimize the data loss and its effects on the data analysis.
3. The application must provide an interface for the developers to create modules that integrate with the application.

The experiments are designed to test the application in (A) a crowded environment with multiple Flow sensors and with different Android OS versions running the application; (B) recording over an extended period to measure battery consumption; (C) user-friendliness on participants; and (D) creating a simple module. When evaluating the experiments, the observations and the results are the most interesting metric as it provides a better perception of whether the design choices (Chapter 4) and implementation (Chapter 5) suffice towards the goal of this thesis. Moreover, the observations and results allow for discussion on improvements which can be made in future work.

6.1 Experiment A: Orchestral Concert to Analyze Musical Absorption using Nidra to Collect Breathing Data

This experiment was conducted in collaboration with master student Joachim Dalgard at the University of Oslo at *RITMO: Centre for Interdisciplinary Studies in Rhythm, Time and Motion*.

The goal of the experiment for Dalgard was to analyze musical absorption, which is a state when individuals allow music to draw them into an emotional experience and becomes unaware of time and space. In order to analyze the effect of musical absorption on individuals, Dalgard gathered 20 participants who were experienced listeners with musical education. The participants attended an orchestral concert by Richard Strauss' Alpine Symphony—a symphonic poem that portrays the experience of eleven hours spent climbing an Alpine mountain—that lasted approximately 50 minutes at Oslo Concert Hall on April 3rd and April 4th, 2019.

With respect to Nidra, the motivation for this experiment can be summarized into: (1) to test the application in a real-life and crowded environment, and analyze whether other mobile devices interfere or obstruct with the signals between the collecting sensors and mobile devices; (2) to test whether the samples gathered are meaningful, in the sense that the application is collecting the samples from the sensors correctly and handling unexpected disconnections; and (3) to test the application on different Android OS versions, and to put the application in the hands of participants.

The participants were divided into two groups to attend the concert on the two dates. Each participant was equipped with a wireless electromyographic sensor from DELSYS in order to measure heart rate, and a Flow sensor to measure respiration during the concert. RITMO had multiple Flow sensors for disposal; however, they had no suitable mobile application that could record data from these sensors. Also, with the equipment they indented to us, they experienced that Flow sensor tended to disconnect every 10-15 minutes resulting in fragmented recordings for a single session. Therefore, they reached out to the *Insitute for Informatics* in hopes of a solution. Our application was a suiting match for both parties, and we could combine the analysis of musical absorbation with the evaluation of Nidra. We arranged for six Android devices and reached out to the participants to bring their Android devices if they had one. Thus, there were ten participants in each group and six assessed devices. As a precaution, we decided to give out the devices to the participants who scored highest on a test performed on beforehand.

During the concert, there were approximately 800 attendees on the first day and approximately 1500 attendees on the second day. We assume that

Model	Samsung Galaxy S9	OnePlus 3T	Google Pixel XL
Operating System	Android 8.0	Android 8.1	Android 9.0 & Android 7.1.2
Chipset	Exynos 9810	Qualcomm MSM8996 Snapdragon 821	Qualcomm MSM8996 Snapdragon 821
CPU	Octa-core	Quad-core	Quad-core
GPU	Mali-G72 MP18	Adreno 530	Adreno 530
RAM	4 GB	6 GB	4 GB
Battery	Li-Ion 3000 mAh	Li-Ion 3400 mAh	Li-Ion 3450 mAh
Bluetooth	5.0, A2DP, LE, aptX	4.2, A2DP, aptX HD, LE	4.2, A2DP, LE, aptX

Table 6.1: Device models used during the concert.

most of the attendees had a mobile device, and probably many of them had BlueTooth activated on the device. As such, we were able to replicate an environment (on a larger scale) where other devices might interfere with the signals between the collecting sensor and the mobile device during recording. Also, we were able to install the application on multiple mobile devices with different Android OS versions and put the application in the hands of the participants.

6.1.1 Preperations

To prepare for the experiments, we configured the applications on the assessed mobile devices and stress-tested the application to prevent any unforeseen events or bugs that could occur during the recording. Also, before the concert, we had to ensure that each participant had the sensors placed correctly on their body and given the correct mobile device and Flow sensor. Below, we describe the preparation in more detail.

Device Configuration The device models in our disposal had to be configured with the applications to enable recording on Nidra. First, the data streams dispatching module was installed on the devices. Second, the sensor wrapper for the Flow sensor was installed on the devices and configured with one Flow sensor—in order to reduce the time to set up the mobile device with a Flow sensor on the participant before the concert. Lastly, the Nidra application was initiated with the id (A–F) of the sensor that the participant should use to keep track of each participant's sensor and device model. In Table 6.1, the device models are listed with their specifications and Android version. In the following list the id is mapped with the device model; the experiment will describe the device model based on the id, respectively: (A, B, C) Samsung Galaxy S9; (D) Google Pixel XL (version: 9.0); (E) Google Pixel XL (version: 7.1.2); and (F) OnePlus 3T.

Body Placement The respiration (breathing) value is based on the participant body circumference, and changes are associated with breathing. The participant was instructed to place the sensor around their thorax (just below the armpits), in order to measure the expansion and contraction of the rib cage.

6.1.2 Results

The records from the various mobile devices were gathered by using the sharing functionality in the application and sent to our application (over e-mail). There were thirteen mobile devices combined for both dates—one of the participants had an Android device—however, the application crashed on one of the mobile devices during the recording. Therefore, we have access to twelve recordings from the concerts.

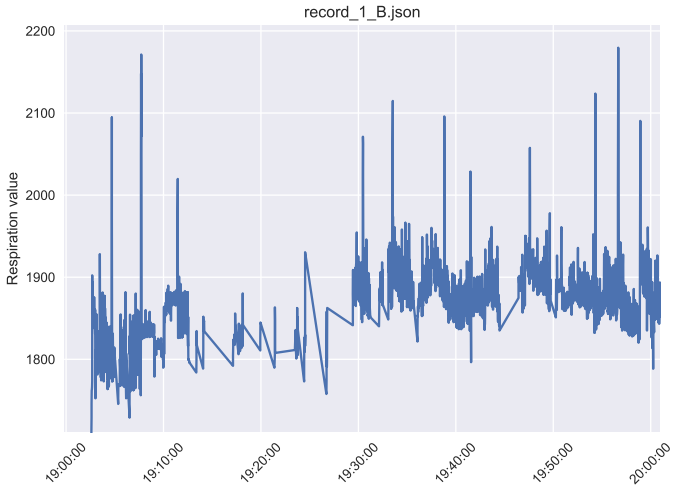

Figure 6.1: Record obtained from the device model B on day 1 of the concert.

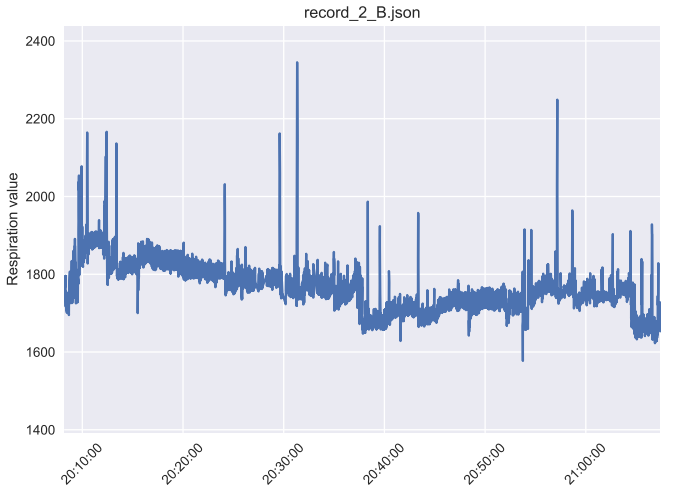

Figure 6.2: Record obtained from the device model B on day 2 of the concert.

Figure 6.1 and Figure 6.2 present two of these twelve recordings (rest can is found in Appendix B) that are of most interest to us in a time-series

graph. The Y-axis represents the respiration value, and the X-axis the time of respiration value acquisition—*day one* of the concert started at the time of 19:00 and ended at 20:00, while *day two* started at the time of 20:10 and ended 21:00.

Moreover, from the thesis by Løberg [35], we can group the signals strengths based on four types of breathing patterns: (1) *normal breathing*—normal exhaling and inhaling (12-18 breaths per minute); (2) *no breathing*—close to flat rates over a long period; (3) *shallow breathing*—rapid inhaling and exhaling; and (4) *deep breathing*—prolonged inhaling or exhaling (also denoted as fluctuations). From the figures, we can see that the respiration value is stable with some fluctuations and disconnections. Disconnections are defined as when the line is sloping or benching in an extended period (e.g., 20 seconds or more), while fluctuations are samples that spikes (deep breathing) in the graph. We should keep in mind that the margin for the normal breathing respiration value can vary throughout the recording based on body or sensor position and movements (e.g., sitting relax or tense on a chair).

Figure 6.1 has many disconnections (further analyzed later) with unstable breathing. There are multiple occurrences of deep breathing throughout the recording, with some shallow breathing at the start and end of the recording. Hypothetically, deep breathing might be a sign of musical absorption; however, we have no concrete analysis of this matter, and it is out of the scope for this experiment motivation. Moving on, Figure 6.2 shows fewer disconnects compared to Figure 6.1, and with much more concise breathing patterns and resemblance to normal breathing. Although, there are noticeably deep breathing throughout the recording, with a few shallow breathings at the beginning. Conclusively, both Figures shows no signs for no breathing during the recording. However, there are signs of normal breathing, with few instances of shallow and deep breathing.

6.1.3 Analysis

In this section, we discuss all twelve records by analyzing the data. It is of special interest to find occurrences of disconnections—when the samples are stagnant in more extended period—and the time the sensor is disconnected during the recording.

Model	Samples Count	Loss Count	Loss Percentage	Disconnection Count	Disconnection Time
A	5145	0	0 %	0	00:00
B	3363	1782	34 %	13	19m:14s
C	4189	956	19 %	7	8m:20s
D	3501	1644	32 %	5	18m:30s
E	5144	1	0.02 %	0	00:00
F	5145	0	0 %	0	00:00

Table 6.2: Day 1—Duration: 1 hour & Expected Sample Count: 5145.

Model	Samples Count	Loss Count	Loss Percentage	Disconnection Count	Disconnection Time
A	4286	2	0.05 %	0	00:00
B	4161	127	3 %	4	1m:25s
C	4286	2	0.05 %	0	00:00
D	2576	1712	40 %	7	24m:35s
E	4285	3	0.06 %	0	00:00
F	4288	0	0 %	0	00:00

Table 6.3: Day 2—Duration: 50 mins. & Expected Sample Count: 4288.

After analyzing the data from the record, Table 6.2 and Table 6.3 presents the six devices for the two dates. Each table exhibits data that is extracted from each record from the mobile device and are characterized as:

Expected Sample Count The expected number of samples that can be acquired in the period of the recording (based on the frequency of sample output by Flow sensor, which is approximately 1.5 Hz).

Sample Count The number of samples that were gathered in the duration of the recording.

Loss Count The number of missing samples based on "expected sample count". This can be calculated as:

$$Loss\ Count = Expected\ Sample\ Count - Samples\ Count \tag{6.1}$$

Loss Percentage The percentage of missing samples based on the expected samples. This can be calculated as:

$$Loss\ Percentage = (1 - \frac{Samples\ Count}{Expected\ Sample\ Count}) * 100 \tag{6.2}$$

Disconnection Count Is the number of disconnections that occurred within the duration of the recording.

Disconnection Time Is the accumulated time of disconnections.

The device models A, E, and F are noticeably accurate (with some noises which presumably occurred during parsing); there are no apparent disconnects during the recording for these models both of the days. However, model B and C show a high loss count on the first day—with a disconnection count of 13 of which the disconnection time is 19 minutes and 14 seconds (34% loss percentage) for model B, and a disconnection count of 7 of which the disconnection time is 8 minutes and 20 seconds (19% loss percentage) for model C—while on the second day, the loss percentage decreased with 31% for model B and 18.95% for model C, resulting in a close to 0% loss percentage for both devices; noticeably, model B shows four disconnects during the recording, however, they only last for 1 minute and 25 seconds in total. Moreover, device model D shows an alarmingly high loss percentage, which is reflected in the disconnection time, both of the days. The mobile device has a loss percentage of 32% on the first day and 40% the second day, that is an increase of 8% of the day before.

6.1.4 Discussion

Based on the analysis, we can see a variety of behavior of disconnections on the mobile devices for the two dates. An assumption is that the mobile device or Flow sensor is malfunctioned; however, we have no data that indicates whether a sensor was connected with the same mobile device for the both dates—we had the data for the first date, however, the sensors and mobile devices were switched at the second date because of unforeseen events—therefore, this assumption is debunked. Moreover, we see that model D (Google Pixel XL, version 9.0) is unstable for both of the days, in contrast to the other device models which ran a lower Android version. However, yet again, we have no accurate data that can prove that the Android version, mobile device, or the Flow sensor is causing the disconnects—albeit, this can be further tested in the lab.

Moreover, the ability of reconnecting with the sensor source on disconnects, shows that data obtained from the recording is more meaningful (worst-case 40% of the data was lossed) on analysis.

6.1.5 Conclusion

To summarize the experiment, we were able to test the application in a real-life and crowded environment. The application managed to record samples that lasted up to 1 hour with the Flow sensor and various device models with different Android OS versions. However, one of the application crashed, and we were unable to find the source of the problem. Based on the samples from the records, it is identified that the Flow sensor has a tendency of disconnecting, but the application managed to reconnect with the sensor during the recording. In the end, the records were successfully shared across applications, which enabled us to analyze the recordings.

To conclude, we evaluate whether the tests sufficed the motivation of the experiment:

1. *test the application in a real-life and crowded environment, and analyze whether other mobile devices interfere or obstruct with the signals between the collecting sensors and mobile devices.*

   While we are able to conduct an experiment which had approximately 800 attendees on the first day and approximately 1500 attendees on the second day of which probably most had their BlueTooth activated, the experiments lack data to support whether interference did affect the sampling from the Flow sensor to a mobile device. However, all of the recordings show at least 60% of the concert was recorded (and some recordings are 100%). To conclude this test, there is insufficient data to show for inference affected the recording; however, were are able to test the application in a real-life and crowded environment.

2. *test whether the samples gathered are meaningful, in the sense that the application is collecting the samples from the sensors correctly and handling unexpected disconnections.*

   As discussed in the analytics section, model device A, E, and F were able to collect data throughout the concert, without any disconnects, for both days. Model B, C, D had disconnects during the recording; however, all managed to reconnect during the recording. As a result, no more than 40% of the recording was lost for both of the days. To conclude this test, the application can successfully reconnect with the connected Flow sensor during the recording, which results in the data is more meaningful for further analysis.

3. *test the application on different Android OS versions, and to put the application in the hands of participants.*

   We had a wide variety of device models with different Android OS versions running the application (Nidra). To conclude this test, the application was able to run all of the Android version, regardless of the device model, successfully. Also, we were able to put the application in the hands of other participants that used the application in order to measure their breathing data during the concert.

   As a side note, a questionnaire was sent to the participants in order to get feedback on the user-friendliness of the application; however, none of them took the time to answer the questions.

6.2 Experiment B: 9-Hours Recording

As part of this thesis is to collect breathing data over en extended period (e.g., during sleep), it is essential to test for battery usage during this period. In most cases, a user connects their device to a power source to charge the device battery. However, in the cases where that is not viable, the battery consumption of the applications might affect the user's perception of Nidra. Therefore, in this experiment, we are checking for the battery consumption on the mobile device, as well as the Flow sensor, to demonstrate that the application is suited for collecting data overnight.

6.2.1 Description

The experiment was conducted on the OnePlus 3T device (specifications in Table 6.1), and we started of by configuring the Flow sensor wrapper to connect with the Flow sensor and pressing the "record" button in Nidra. Both the device and sensor started with 100% battery level. Also, the device was disconnected from WiFi and left stationary and unattended during the period of recording.

A technical description of collecting, the data from the Flow sensor collects data on 10Hz; however, it sends a packet of data at approximately 1.5 Hz. The data packet is processed by the Flow sensor wrapper and sent to the data streams dispatching module (DSDM). The DSDM has a list of subscribing applications, and distribute the data packet to all of the subscribing application. In our case, the Nidra application has subscribed for the Flow sensor packets. In Nidra, the packet is received, unpacked and immediately stored in a database as a sample entity, with an identifier of the recording session. This process continues until the recording has terminated based on the user's actions.

6.2.2 Results

We calculate the energy consumption—which is measured in milliampere-hour (mAh)—and estimate the average energy consumption during the experiment.

$$Battery\ Consumption = \frac{(start\% - end\%) * device\ capacity\ (mAh)}{recording\ duration\ (hours)} \tag{6.3}$$

Based on the difference of start mAh and end mAh, multiplied by the device battery capacity and divided by the duration of recording, the energy consumption per hour is calculated.

The application collected 45468 samples from the Flow sensor over the course of 9 hours. That means, the sensor sent data packets on a frequency of approximately 1.5 Hz, which is accurate with previous tests. The mobile device had a battery level of *59%* and the Flow sensor had a battery level of *94%* at the end of the recording. Using Equation 6.3, we can calculate the average mAh consumed during 9 hours of recording.

$$\frac{0.41 * 3400}{9} = 154.89\ mAh \tag{6.4}$$

As a result, the device uses approximately 155 mAh. This means that the total battery consumption over the course of the recording was 1395 mAh. Moreover, the device could have been continuing recording for approximately 13 hours more, before running out of battery. Keeping in consideration that the mobile device has an average battery capacity, with a degraded battery capacity due to age, the results seem adequate. However, to compare the results with another application with the same criteria (the same device that is stationary with WiFi turned off), we tested the Spotify application—an application for streaming music—by downloading a song and allowing it to play in the background.

$$\frac{0.025 * 3400}{1} = 85\ mAh \tag{6.5}$$

Using Equation 6.3, we have a rough estimate of the battery usage for the Spotify application. The application drained close to 3% of the battery percentage during 1 hour of streaming; resulting in an approximately 85 mAh consumption.

6.2.3 Discussion

The results show that the process of recording consumes close to two times more battery in comparison with listing to music in the background. While the Spotify application plays a downloaded song in the background, the Nidra application operates with two other applications that forward data collected over BlueTooth in the background. As such, the amount of battery consumption is reflected in the process of recording.

A proposition to reduce the amount of background processing in order to preserve the battery capacity of the mobile device, is to extend the sensor wrapper with a temporary cache or data storage. As such, the data packets are stored in the sensor wrapper during the recording and sent to the subscribing application when the recording is stopped. However, this complicates the structure of a sensor wrapper and might not be feasible in all cases (e.g., the real-time graph becomes obsolete). Although, it can be part of a power-saving configuration in the future.

6.2.4 Conclusion

To summarize, in this experiment, we used the recording functionality in Nidra. To enable this functionality, Nidra notifies the Flow sensor wrapper through the data streams dispatching module to start the data acquisition. The Flow sensor samples data on a frequency of approximately 1.5Hz and the data are forwarded from the sensor wrapper to the data streams dispatching module, which further distributes the data packets to the subscribed applications. Nidra processes the data packets by unpacking and storing them respectively in the database. At the end of the 9-hours recording, the application collected 45468 samples. As a result, the mobile device used *41%* of the battery (1395 mAh) and the Flow sensor used *6%*. Moreover, the device had the possibility of recording for an extra 13 hours.

To conclude, the results of the experiments show that the requirements for recording over an extended period are satisfied. The application can collect data from the Flow sensor, and it is sufficient enough to do it for an extended period. The energy consumption might seem high; however, that is reflected by three applications running simultaneously in the background and data being collected over BlueTooth. Nonetheless, the experiment was successfully conducted, with no apparent form for disconnects or abruptions during the recording.

6.3 Experiment C: Performing User-Tests

One goal of this thesis is that it should be user-friendly for patients to use and understand the functionality of Nidra. To evaluate whether this goal has been achieved, we found two participants that agreed to partake in

the experiment. One of these participants was not proficient in the use of modern technology, while the other participant was tech-savvy[1]. For simplicity, we refer to the participant as participant `A` and `B`, respectively.

The process of this experiment consisted of three parts: a presentation of the application; testing the function of recording, sharing, analyzing; and a survey to evaluate the experience.

Presentation We described the functionality of the application to the participants simply and intuitively—without showing the application. That included the explanation of the recording functionality, possibilities of viewing the data in a graph, and the methods of sharing the recording across applications. The presentation took approximately 10 minutes, including questions and clarifications.

Tests The tests were mainly designed to evaluate *comprehensibility* and *usability* in the application. Comprehensibility is to evaluate the participants' ability to understand the functionality of the application, while usability to measure ease of use of the application.

Survey The participants were instructed to fill out a questionnaire after these tests were conducted—such that the participant had their feedback fresh in memory. We structured the questions in the questionnaire accordingly to the PACMAD model [29].

6.3.1 Testing

To evaluate whether the application presents the functionality adequately and suffice the goal of the tests, we tested the participants with certain tasks:

T1 Proceed to navigate in the application and start a recording.

T2 Find the interface to view statistics in order to check for connected sensor sources and the graph for the sampling.

T3 Stop the recording and give the record a name, description, and rating.

T4 Find the record in the feed of recordings.

T5 Share the record over e-email.

T6 View the analytics for the record and interacting with the graph (e.g., zooming and scrolling).

For this test to be successful, the participants must for all of the tasks be able to identify the actions correctly. These tests were designed to observe the process of recording, sharing, and analyzing. Thus, the functionality of modules were left behind.

[1] well proficient in the use of modern technology.

6.3.2 Observations

Participants A—less technical—had to familiarize with the setting of the application, thus had a difficult start. It took the participant longer than expected to perform T1, however, managed to proceed on the recording. The participant was a bit uncertain whether the recording had started or not. However, the participant was flabbergasted by the rippling effect the recording screen presented. The T2 was somewhat tedious to perform because the participant tried to click on the interface where it says it should be expanded (by swiping). Also, the graph was a bit hard to interact with because the interface above the graph kept moving while performing interactions with the graph. T3 and T4 went fine, the participant filled out the title, description, and a rating and saved the recording, and found the recording on the feed screen. However, T5 and T6 were a bit unclear to the participant, because the records in the list are collaped, and in order to find the actions you have to press the record to expand the information. However, after figuring out how to expand the records, the participant managed to perform both tasks sufficiently.

Participants B outperformed participant A, due to the participant was more familiar with technical systems. The participant managed to start the recording quickly, however, was unsure whether the recording had started (clicking around on the interface for feedback) until the ripple effects appeared on the screen. Similar to participant A, participant B clicked on the interface for statistics, rather than swiping, and had a hard time to interact with the graph due to the interface moving. The participant B managed to perform T3–T6 with ease and without any interrupts or objections.

6.3.3 Survey

A questionnaire was filled out by the patients after the tests. The questions followed the PACMAD model [29], which is a model to identify the usability attributes and are structured into: effectiveness, efficiency, satisfaction, learnability, memorability, errors, and cognitive load. We created a survey based on some of the structure, as we were unable to perform any cognitive load or memorability tasks during the testing. Below, the questions from the patients are listed—where 1 is very-hard/very-bad/not-satisfied, and 5 is very-good/very-easy/very-satisfied:

Effectiveness & Efficency

- What were your initial thoughts on the application

  Participent A: It looked nice and simple.

  Participent B: The application seemed very modern and elegant.

- How difficult was it to start a recording

Participent A: 3

Participent B: 4

- How would you rate the feedback you got during a recording

  Participent A: 2

  Participent B: 3

- How difficult was it to stop a recording?

  Participent A: 5

  Participent B: 5

- How difficult was it to browse/find previous recordings?

  Participent A: 4

  Participent B: 5

- Did you have any encounters were the application did not supply you with enough information?

  Participent A: On the recording screen.

  Participent B: The recording screen could have been more informing, with more text or an introduction of how the process of recording works. Besides this, everything worked fine.

Satisfaction

- How satisfied were you with the "journey"?

  Participent A: 4

  Participent B: 4

- How satisfied were you with this application overall?

  Participent A: 4

  Participent B: 4

Errors

- If you encountered any crashes or errors during the time you used the application, please answer the question below.

  Participent A: None.

  Participent B: None.

Feedback and Improvements

- How user friendly did you find the application to be?

  Participent A: 4

  Participent B: 4

- How would you rate the color palette of the application?

  Participent A: 5

  Participent B: 5

- How would you rate the general layout of the application (buttons, text, navigation, etc...)?

  Participent A: 5

  Participent B: 5

- Do you have any feedback/improvements to the application itself?

  Participent A: No.

  Participent B: Nothing more than I described earlier.

6.3.4 Discussion

As for the observations, the participant managed to perform most of the tasks, albeit the task T1 and T2 were hard to comprehend. Both participants had difficulties in understanding whether the sensor was collecting data, despite the ripple-effects on the screen. The source of this problem is that the sensor some times takes up to 30-60 seconds to start collecting data, making the user wait on the ripple-effect that indicates that the recording has started. Arguably, the user can familiarise with the state of the recording and the ripple-effects; however, to a new user that is not feasible. Also, the interface for statistics was a bit tedious to work with, mainly due to the swiping effect to show the interface and the interactions with the graph making the interface move. Besides this, there were no noticeable complains by the participant.

As for the survey, the observations reflect the answers in the questionnaire. Most of the poor feedback was directed towards the recording screen being less informative than expected. They found the color scheme of the application to be smooth and fitting, and the application to be modern. Also, they found the general layout to be well organized and overall the application to be user-friendly.

A proposition to the complains of the recording screen is to enlargen the recording button and provide more informative text on the recording screen. As briefly discussed, a possibility is to create an introduction screen, that showcases the functionality of Nidra, the first time the user starts the application. Moreover, the statistics interface can be moved into a separate screen, to preventing the interface from moving while interacting with the graph.

6.3.5 Conlusion

To summarize, we conducted an experiment on two participants with a predefined set of tasks in order to measure the user-friendliness of the application (Nidra). Most of the tasks were regarding the functionality of recording, sharing, and analyzing. The participants were presented with a brief overview of the functionality of the application before starting the testing. The tests were mainly designed to evaluate the comprehensibility and usability of the application. During the testing, we observed the interactions the participants made for each task and followed up with a survey for them to answer. Based on this, we gained a broader understanding of the user-friendliness of the application.

To conclude, the tests were created to measure the comprehensibility and the useability. Overall, the application suffices these measurements; however, the recording screen can be improved in the future. In our perspective, the amount of action that can be performed in that particular screen is limited to three actions (e.g., starting the screen, viewing the statistics, and stopping the recording). However, the participants were not familiarized with the setting of the recording functionality, in which we could have introduced the functionality of the recording even further (e.g., with more text or an introduction before the recording screen). Besides this functionality, the participants found the rest of the application to be user-friendly and easy to use.

6.4 Experiment D: Creating a Simple Module

One of the requirements in this thesis is to provide an interface for the developers to create modules, which allows for data enrichment and extended functionality in Nidra. In order to test for this, we found one participant that had experience in software development, and with some experience in Android development.

The tests consisted of creating a new module that utilizes the records from Nidra, and of finding the record with the highest number of samples—it was sufficient to display the correct answer on the screen (the development of a user-interface was not evaluated). Based on these tests, we evaluate the procedure and the difficulties of creating a new module.

Before the participant started on creating a new module, we introduced the concepts and data structure of Nidra. Mainly, that Nidra formats the all of the data (e.g., records and corresponding samples) into a JSON string, and the JSON string is put into a bundle with the key *data*, and sent upon launch of the module-application.

6.4.1 Observations

Although the first task seems intuitive, the participant had a rough start. The participant was aware that module-application received data in a bundle, and the data extraction could be performed by specifying the key. However, for each time the participant compiled the module-application crashed—that is because the bundle is empty if the module-application is launched directly, and not through the Nidra application (which supplies the module with the data). However, that was corrected quite quickly.

The next challenge for the participant was to understand how to decode the JSON string into valid objects in Java. The participant studied the structure of the string received on the launch and managed to come up with a solution. First, the participant created an object which encapsulates the record and a list of samples. Then, creating three separate objects (i.e., record, sample, and user) that were identical to the data structure. In the end, the participant could decode the JSON string into a list and retrieve the necessary data.

Once the participant had all of the data, the final task was easily accomplished. The participant iterated through the list of records, and found the one with the highest sample count, and displayed it on the screen.

6.4.2 Results

At the time of the experiment, there were five records on the mobile device, and the highest was the record named *Record 17* with *163 samples*. The participant's module displays the same information. Thus, the experiment was successful conducted. It took the participant approximately 20 minutes to develop this module-application, where most of the time went on parsing the data.

6.4.3 Discussion

The participant managed to create a module with some hurdles in the way. The first noticeable occurrence was after compiling the application, the participant had to open its module through Nidra in order to get the data. To overcome this, there are two ways to improve the relationship between Nidra and a module. One way is to establish an IPC connection (discussed in the design chapter) with the use of a binder, and send the data from Nidra to the application through the IPC connection. The second way is to cache the data in the module-application for the temporary storage of the data. However, both methods increase the complexity of module-application and Nidra.

The second noticeable occurrence was the parsing of the JSON data, as JSON is a bit tedious to work with in Java. As of now, there is no direct support for parsing JSON in Java; hence, third-party libraries have to be used in order to do so.

6.4.4 Conclusion

To conclude this experiment, the participant was able to create a new module that used the data from Nidra to display the record with the highest sample count. As such, the facilitation of modules allows future developers to create modules, without having to understand how Nidra operates—besides the data structure and how to receive the data—to create modules that can extend the functionality of Nidra.

Moreover, as a result of the observation of the time spent initializing a module-application, we have included a template code (found in Appendix C) to make module implementation easier for future developers.

6.5 Summary of Results

In this chapter, we conducted various experiments with the application (Nidra). The experiments consisted of testing the application in a (A) real-life and crowded environment, installing the application on various device models running different Android OS versions, and checking whether the application managed to reconnect upon disconnections; (B) recording over an extended period to measure the battery consumption; (C) user-testing of the application; and (D) creating a simple module. Moreover, the experiments were constructed in order to evaluate the system requirements, which we discuss in Section 6.6; however, in this section, we reinstate the main findings from the results of the experiments.

As for experiment A, we were able to test the application in a real-life and crowded environment with the opportunity of attending a concert that had approximately 800 attendees the first day 1500 attendees the second day. As for the result of the experiment, we were able to test the application on various mobile devices running different Android version. Some of the mobile devices were able to collect all of the breathing data throughout the concert, while others disconnected during recording. However, Nidra was able to reconnect with the disconnected sensor sources, resulting in that the worst-case of sample loss was no more than 40%. As such, the functionality of reconnection with the sensor source on disconnects allows for more meaningful analysis of the data.

As for experiment B, the application was tested for battery consumption over the period of 9-hours of recording. To enable the recording functionality, the application (Nidra) operates with two other applications (DSDM and Flow Sensor Wrapper) that run in the background, in which

the Flow sensor transmits data over BlueTooth. As for the results of the experiment, the recording process consumes approximately *155 mAh*; thus, after 9-hours of recording the mobile device has consumed *1395 mAh* of its battery capacity. The average battery capacity of low-end mobile devices is approximately 2000 mAh. As such, the user has enough battery to operate the mobile device after 9-hours of recording.

As for experiment C, we evaluated the user-friendliness of the application based on comprehensibility and usability. In the experiment, we had two participants with different knowledge and perspective of technology, that tested the application while we observed the interactions. As for the results of the experiment, the participants managed to complete all of the tasks that were defined, sufficiently; however, some minuscule complications made some of the tasks harder to comprehend. Overall, the participants found the application to be user-friendly and easy to use.

As for experiment D, we allowed a participant to create a module application that used the records from Nidra in order to find the record with the highest sample count. As for the results of the experiment, the participant managed to create a module that fulfilled the goals; however, the time that took parsing the data, lead us to create a template (found in Appendix C), which expedite the creation of future modules.

6.6 Concluding Remarks

In this chapter, we conducted experiments to evaluate the system requirements, which contributes towards fulfilling the goals of this thesis. Below, we describe the requirements and discuss the experiments which have contributed to fulfilling them.

1. *The application must provide an interface for the patient to (i) record physiological signals (e.g., breathing data); (ii) present the results; and (iii) share the results.*

   This requirement is supported by experiments A, B, and C that proves that the application is capable of (i) recording breathing data over an extended period of time using the Flow sensor; (ii) present the results in a time-series graph that allows analysis of the records; (iii) sharing the records from various mobile devices over a media (i.e., e-mail).

2. *The application must ensure that upon sensor disconnections, the connectivity is reinstated to minimize the data loss and its effects on the data analysis.*

   This requirement is supported by experiment A that proves that the application managed to reconnect with the sensor source upon disconnections. As a result, the analysis performed on the records allowed for a deeper understanding of the breathing patterns of a user.

3. *The application must provide an interface for the developers to create modules that integrate with the application.*

   This requirement is supported by the experiment D that proves that other developers can make modules and add them in Nidra. This allows future developers to create modules that can perform more advanced analysis on the data (e.g., machine learning techniques) or extend the functionality of Nidra.

Besides few subtle improvements that can be made to Nidra, the system requirements suffice towards the goal of making an application that can record, share, and analyze breathing data with the Flow sensor over an extended period. Also, the possibilities for future developers to create independent applications which extend the functionality of Nidra.

Chapter 7

Conclusion

7.1 Summary

Our motivation for this thesis was to extend the CESAR project by creating an application for patients to be able to record, share, and analyze breathing data from home. The purpose of the data is to aid researchers/doctors in analysis, diagnosis, and examination of the patient for sleep-related breathing disorders (e.g., obstructive sleep apnea). However, the application can be applied to other fields of study, such as recording breathing during physical activity. Moreover, the application should facilitate an interface for developers to create modules which extend the functionality or enrich the data of the application.

To achieve this, we designed an application by separating the tasks[1] into concerns, for each of which a structure consisting of components with several functionalities and design choices were proposed. Based on the design, we implemented an application in Android, called Nidra. We created a user-friendly interface for the users to record breathing data over an extended period (e.g., overnight) by creating a sensor wrapper that supports the collecting of breathing data from the Flow sensor over the BlueTooth LE protocol with the data streams dispatching tool. The Flow sensor was prone to disconnecting; thus, we incorporated a mechanism which reconnects with the Flow sensor upon disconnects. As a result, the data is richer and more meaningful upon analysis. Further, we added the support for sharing the records—containing the samples gathered over the period of recording—across applications, allowing the patients to share their records with the researchers/doctors over a media (e.g., e-mail). Finally, we integrated a simple form for analysis of the data within the application; in other words, a time-series graph with the breathing value and time of acquisition. Additionally, we also created an interface for developers to create modules which allow for extended functionality and data enrichment. For example, a module can use the data from patient

[1] recording, sharing, analyzing, modules, storage, and presentation

records in order to predict sleep apnea with the use of machine-learning techniques.

To evaluate the application, we performed various experiments including recording in a crowded environment with various mobile device running different Android OS versions, over an extended period while measuring the battery consumption, evaluating the user-friendliness of the application, and the process of creating a new module. Based on the results, we see that the application is capable of collecting breathing data from various mobile devices—running different Android OS version—using the Flow sensor. Also, our application is capable of reconnecting with the Flow sensor upon disconnects; from the experiment, the worst-case of data loss was no more than 40%. Moreover, we tested the battery consumption of the application. Keeping in consideration that there are three other applications, including our application, running in the background while collecting data over BlueTooth LE protocol, the results show that the process of recording for 9-hours uses 1395 mAh of the battery capacity. That is reasonable in regards to the average battery capacity of mobile devices. Next, we evaluated the user-friendliness of the application on the functionality of recording, sharing, and analyzing. While overall, our application is user-friendly and easy-to-use, the recording screen seems to lack information to guide new users in the process; however, that can easily be adjusted. Finally, we allowed a developer to create a module that uses the data from our application in order to find the record with the highest sample count. As a result, the module worked as intended, and it successfully extended the functionality of Nidra.

7.2 Contributions

As defined in our problem statement we set three research goals, the following section reinstate and describe how this thesis contributes to them.

Goal 1 *Integrate the support for Flow sensor with the extensible data streams dispatching tool.*

This goal is supported by the development of the Flow sensor wrapper, which integrates with the data streams dispatching tool. The Flow sensor uses BlueTooth LE as a communication protocol. As such, we used the APIs provided by Android to connect with the sensor, and focused on extracting the flow (breathing) data and battery level, as well as the MAC address and firmware level. By creating a sensor wrapper for the Flow sensor and integrating with the data streams dispatching tool, we extend the tool for integration of the Flow sensor. That allows future developers to operate with the Flow sensor in their analysis.

Goal 2 *Research and develop a user-friendly application which facilitates collec-*

tion of breathing data with the use of the extensible data streams dispatching tool.

This goal is supported by the development of the Nidra, which provides an interface for recording, sharing, analyzing breathing data collected with the Flow sensor using the data streams dispatching tool on a mobile device. With this application, we allow patients to record their breathing data over an extended period, and to analyze the data within the application. Also, we allow the patients to send their record—e.g., over e-mail—to their researchers/doctors for further analysis. As such, we provide an interface to the patients to monitor their breathing from home. However, this application can also be used in other fields of study (e.g., physical activity), as shown in the experiements we collected breathing data in order to analyze the effects of music absorption on indivuals.

Goal 3 *Create an extensible solution such that the developers can integrate stand-alone applications in our application.*

This goal is supported by the integration of modules in the Nidra application. We allow future developers to create and integrate their applications functionality into Nidra, without having to understand how Nidra operates, and leverge the data Nidra provides. This allows future developers to enrich the data or extend the functionality in Nidra, resulting in patients only having one application for the detection, analysis, examination of sleep apnea.

7.3 Future Work

Over the course of this thesis, we were able observe enhancements and improvements that can be made to the application. While the application fulfilles the goals defined in the problem statement, there are possibilities to make the experience of the application even richer. Below, we present the a short description of future work alongside with our proposition.

Improve the user-friendliness of Nidra: In retrospect, we could see that the participants had trouble with finding the record button, as well as the indication of whether a recording had started. A proposition is to enlargen til record button, to make it more visible to the users. For the recording, perhaps have a more informative description of the various states (e.g., connecting, recording, or disconnected).

Add support for other physiological data The main focus in this thesis, was to gather breathing data during sleep. However, the possibility of extending Nidra to support other physiological data is possible (e.g., heart rate). The extensible approach to application, allows future developers to integrate the support for these data types, by simply extending the database and recording logic.

Create an interface for sharing A proposition in the design for sharing, was to create an interface solely for sharing data between users, without accessing other applications (e.g., mail). The idea was to create a server that maintains a user-base of patients and researchers/doctors, and a repository for shared records. By providing an interface for the patient to select the desired recipient, would allow for a simpler and convenient sharing for both the patient and the research/doctor.

Bi-directional channel between Nidra and modules As of now, the data is packed into a JSON string and bundled into an Intent on launch. As discussed in the design, this would mean that for the module to obtain the data the user has to launch the module through the application. For simplicity, this is sufficient; however, for future modules that depends on analyzing the data in real-time, that is not a suitable solution. Therefore, by establishing a bi-directional channel with Nidra and the modules is a solution to this problem. By utilizing binder's (with AIDL) for IPC, the data flow can occur both ways. Nidra can obtain reports or results from the modules, and the modules can obtain samples in real-time and/or selective desired records, respectively.

Filter modules based on package-name Currently, all of the installed applications are listed when selecting a new module to add in Nidra. To improve the user experience, we can have that modules prerequisite is that the package name should start with *com.cesar.X*—or something similar. With this, we can filter out unwanted applications, and display the module-applications that are eligible to the user.

Bibliography

[1] Ecma International 2017. *The JSON Data Interchange Syntax*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 01/06/2019).

[2] *Activities*. Android Developer. URL: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/activities.html (visited on 23/06/2019).

[3] S. Alqassim et al. 'Sleep Apnea Monitoring using mobile phones'. In: *2012 IEEE 14th International Conference on e-Health Networking, Applications and Services (Healthcom)*. Oct. 2012, pp. 443–446. DOI: 10.1109/HealthCom.2012.6379457. (Visited on 26/06/2019).

[4] *Android based eHealth applications with BiTalino sensors*. University of Oslo: Institute for Informatics. URL: http://www.mn.uio.no/ifi/studier/masteroppgaver/dmms/android-based-ehealth-applications-with-bitalino-s.html (visited on 05/05/2018).

[5] *Android Graph Library*. Github Inc. URL: https://github.com/jjoe64/GraphView (visited on 20/03/2019).

[6] *Android Interface Definition Language (AIDL)*. Android Developer. URL: https://developer.android.com/guide/components/aidl (visited on 24/06/2019).

[7] *Application Fundamentals*. Android Developer. URL: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/fundamentals.html (visited on 23/06/2019).

[8] *Assigned Numbers*. Bluetooth Technology. URL: https://www.bluetooth.com/specifications/assigned-numbers/ (visited on 25/06/2019).

[9] *Background Execution Limits*. Android Developers. URL: https://developer.android.com/about/versions/oreo/background#broadcasts (visited on 10/07/2019).

[10] *Binder*. Android Developer. URL: https://developer.android.com/reference/android/os/Binder (visited on 24/06/2019).

[11] *Bluetooth low energy overview*. Android Developer. URL: https://developer.android.com/guide/topics/connectivity/bluetooth-le (visited on 25/06/2019).

[12] Daniel Bugajski. 'Extensible data streams dispatching tool for Android'. MA thesis. University of Oslo, 2017. URL: https://www.duo.uio.no/handle/10852/60350.

[13] *CESAR: Using Complex Event Processing for Low-threshold and Non-intrusive Sleep Apnea Monitoring at Home*. UiO: Department of Informatics. URL: https://www.mn.uio.no/ifi/english/research/projects/cesar (visited on 21/06/2019).

[14] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: https://www.w3.org/TR/REC-xml/ (visited on 01/06/2019).

[15] *Content Providers*. Android Developer. URL: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/providers/content-providers.html (visited on 23/06/2019).

[16] Abe Crystal and Beth Ellington. 'Task analysis and human-computer interaction: approaches, techniques, and levels of analysis'. In: *AMCIS*. 2004, pp. 2–3. URL: https://pdfs.semanticscholar.org/fbd2/b61c998cb3ad8427759a45370a02d9338c31.pdf (visited on 02/05/2019).

[17] *Dark theme*. Material Design. URL: https://material.io/design/color/dark-theme.html#usage (visited on 04/06/2019).

[18] *Data and file storage overview*. Android Developer. URL: https://developer.android.com/guide/topics/data/data-storage (visited on 25/06/2019).

[19] *Data and file storage overview*. Android Developers. URL: https://developer.android.com/guide/topics/data/data-storage (visited on 05/06/2019).

[20] *Fact check: Is smartphone battery capacity growing or staying the same?* Android Authority. URL: https://www.androidauthority.com/smartphone-battery-capacity-887305/ (visited on 30/05/2019).

[21] *FileProvider*. Android Developer. URL: https://developer.android.com/reference/android/support/v4/content/FileProvider (visited on 23/06/2019).

[22] *Fragments*. Android Developer. URL: https://developer.android.com/guide/components/fragments (visited on 23/06/2019).

[23] R. Fu, Z. Zhang and L. Li. 'Using LSTM and GRU neural network methods for traffic flow prediction'. In: *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. Nov. 2016, pp. 324–328. DOI: 10.1109/YAC.2016.7804912. (Visited on 07/06/2019).

[24] H Gray Funkhouser. 'Historical development of the graphical representation of statistical data'. In: *Osiris* 3 (1937), pp. 269–404. URL: https://www.journals.uchicago.edu/doi/pdfplus/10.1086/368480 (visited on 07/06/2019).

[25] Svein Petter Gjøby. 'Extensible data acquisition tool for Android'. MA thesis. University of Oslo, 2016. URL: https://www.duo.uio.no/handle/10852/53004.

[26] Robert L. Glass. 'A structure-based critique of contemporary computing research'. In: *Journal of Systems and Software* 28.1 (Jan. 1995), pp. 3–7. ISSN: 0164-1212. DOI: 10.1016/0164-1212(94)00077-z. URL: http://dx.doi.org/10.1016/0164-1212(94)00077-z (visited on 15/07/2019).

[27] *Guide to app architecture*. Android Developer. URL: https://developer.android.com/jetpack/docs/guide (visited on 20/06/2019).

[28] *Intents and Intent Filters*. Android Developer. URL: https://developer.android.com/guide/components/intents-filters (visited on 24/06/2019).

[29] Roesnita Ismail, Norasikin Fabil and Ashraf Saleh. 'Extension of pacmad model for usability evaluation metrics using goal question metrics (Gqm) approach'. In: *Journal of Theoretical and Applied Information Technology* 79 (Sept. 2015). URL: https://www.researchgate.net/publication/325484529_Extension_of_pacmad_model_for_usability_evaluation_metrics_using_goal_question_metrics_Gqm_approach (visited on 05/07/2019).

[30] *Keep the device awake*. Android Developer. URL: https://developer.android.com/training/scheduling/wakelock (visited on 25/06/2019).

[31] Santosh Kumar et al. 'Mobile health technology evaluation: the mHealth evidence workshop'. In: *American journal of preventive medicine* 45.2 (2013), pp. 228–236. DOI: 10.1016/j.amepre.2013.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0749379713002778 (visited on 09/07/2019).

[32] Walter L. Hursch and Cristina Videira Lopes. 'Separation of Concerns'. In: (Mar. 1995), pp. 3, 16. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=031E83D200FD8770C255B48EA0C2E1C2?doi=10.1.1.29.5223&rep=rep1&type=pdf (visited on 24/05/2019).

[33] Huoran Li, Xuanzhe Liu and Qiaozhu Mei. 'Predicting Smartphone Battery Life based on Comprehensive and Real-time Usage Data'. In: (Jan. 2018), p. 2. URL: https://www.researchgate.net/publication/322498404_Predicting_Smartphone_Battery_Life_based_on_Comprehensive_and_Real-time_Usage_Data (visited on 30/05/2019).

[34] *LiveData Overview*. Android Developer. URL: https://developer.android.com/topic/libraries/architecture/livedata (visited on 20/06/2019).

[35] Fredrik Løbberg. 'Measuring the Signal Quality of Respiratory Effort Sensors for Sleep Apnea Monitoring: A Metric Based Approach'. MA thesis. University of Oslo, 2018. URL: https://www.duo.uio.no/handle/10852/62777.

[36] Tian Lou. 'A Comparison of Android Native App Architecture – MVC, MVP and MVVM'. In: (), p. 57. URL: https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf (visited on 20/06/2019).

[37] Innocent Mapanga and Prudence Kadebu. 'Database Management Systems: A NoSQL Analysis'. In: *International Journal of Modern Communication Technologies And Research (IJMCTR)* Volume-1 (Sept. 2013), pp. 12–18. URL: https://www.researchgate.net/publication/258328266_Database_Management_Systems_A_NoSQL_Analysis (visited on 15/05/2019).

[38] Stephen McGrath and Jonathan Whitty. 'Stakeholder defined'. In: *International Journal of Managing Projects in Business* 10 (July 2017), pp. 4, 13, 14. DOI: 10.1108/IJMPB-12-2016-0097. (Visited on 01/05/2019).

[39] *Municipal health care service*. Statistics Norway. URL: https://www.ssb.no/en/helse/statistikker/helsetjko (visited on 19/07/2018).

[40] Rajalakshmi Nandakumar, Shyamnath Gollakota and Nathaniel Watson. 'Contactless Sleep Apnea Detection on Smartphones'. In: *GetMobile: Mobile Computing and Communications* 19 (Dec. 2015), pp. 22–24. DOI: 10.1145/2867070.2867078. (Visited on 26/06/2019).

[41] T Penzel et al. 'The use of a mobile sleep laboratory in diagnosing sleep-related breathing disorders'. In: *Journal of medical engineering & technology* 13.1-2 (1989), pp. 100–103. URL: https://www.tandfonline.com/doi/pdf/10.3109/03091908909030206 (visited on 26/06/2019).

[42] *Platform Architecture*. Android Developer. URL: https://developer.android.com/guide/platform (visited on 21/06/2019).

[43] *Power Management*. Android Developer. URL: https://source.android.com/devices/tech/power/mgmt (visited on 25/06/2019).

[44] *Processes and Threads*. Android Developer. URL: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/processes-and-threads.html#IPC (visited on 24/06/2019).

[45] *Save data in a local database using Room*. Android Developer. URL: https://developer.android.com/training/data-storage/room/index (visited on 20/06/2019).

[46] *Secure an Android Device*. Android. URL: https://source.android.com/security (visited on 30/05/2019).

[47] *Services*. Android Developer. URL: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/services.html (visited on 23/06/2019).

[48] *Sleep Disorders: Sleep-Related Breathing Disorders*. PubMed. URL: https://www.ncbi.nlm.nih.gov/pubmed/28845957 (visited on 19/07/2019).

[49] Audie Sumaray and S. Kami Makki. 'A comparison of data serialization formats for optimal efficiency on a mobile platform'. In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12*. the 6th International Conference. Kuala Lumpur, Malaysia: ACM Press, 2012, p. 1. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184810. URL: http://dl.acm.org/citation.cfm?doid=2184751.2184810 (visited on 26/05/2019).

[50] *SweetZpot FlowTM Sensor*. SweetZpot. URL: https://www.sweetzpot.com/flow (visited on 28/05/2018).

[51] *The color system*. Material Design. URL: https://material.io/design/color/the-color-system.html (visited on 04/06/2019).

[52] M. U.Farooq et al. 'A Critical Analysis on the Security Concerns of Internet of Things (IoT)'. In: *International Journal of Computer Applications* 111.7 (18th Feb. 2015), pp. 1–6. ISSN: 09758887. DOI: 10.5120/19547-1280. URL: http://research.ijcaonline.org/volume111/number7/pxc3901280.pdf (visited on 30/05/2019).

[53] Saurabh Zunke and Veronica D'Souza. 'JSON vs XML: A Comparative Performance Analysis of Data Exchange Formats'. In: 3.4 (2014), pp. 2–4. URL: http://ijcsn.org/IJCSN-2014/3-4%20JSON-vs-XML-A-Comparative-Performance-Analysis-of-Data-Exchange-Formats.pdf (visited on 30/05/2019).

Appendices

Appendix A

Source Code

The source code for the various applications in the context of this thesis is found at following Github repository: *https://github.com/riatio/master*

A.1 File and Folder Structure

Nidra encompasses the implementation performed in Chapter 5. The application follows the MVVM architecture pattern; thus, the naming scheme of the folders advertently follows the separation of the components in the architecture pattern. The folder structure for the application code is separated into:

- **Dispatcher**: contains the code for communication with the DataStreamsDispatchingModule, including the service for data acquisition, and the code for re-connectivity with the sensor sources on sensor disconnect or human disruption.
- **Model**: contains the model for data entities, structure for the sensor data and Flow payload data, and the interface for establishing a connection with SQLite (with Android Room).
- **Utils**: comprises of miscellaneous code ranging from functionality to extract Flow payload to the logic behind the export functionality.
- **View**: include the activities with separation of various views (e.g., feed, module, and recording).
- **ViewModel**: exposes the operations that can be performed on the data entities.

DataStreamsDispatchingModule encompasses the implementation made by Bugasjki. However, the application was extended during the course of this thesis and can be reflected in the following files:

- **SensorDiscovery** listens for broadcasts sent by the sensor wrappers on start. The data passed alongside the broadcasts is extracted of name and package name, and stored in a `SharedPreferences` if it does not exists.
- **Sensor** is the sensor object that is stored into the a SharedPreferences, with the name and the package name of the sensor wrapper.

Flow encompasses the driver for creating a sensor wrapper made by Gjøby [25]. The sensor wrapper adds the support for Flow sensor, and the extension made to enable the sensor wrapper (besides the components introduced by Gjøby) are the following:

- **Bluetooth** include the code for connecting with the Flow sensor with Bluetooth LE. Most of the logic is in the file `BluetoothHandler`, and the code is inspired by the application *RawDataMonitor* sent to us from Sagar Sen at SweetZpot Inc.
- **View** contains the activity for listing the available sensors on the screen.

TestModule encompasses a boilerplate code for creating a new module (further discussed in Appendix C).

Thesis encompasses the LaTeX for this thesis, including figures and UiO master's thesis format.

Graph encompasses the data acquired from the various mobile devices during the two concert dates, including a Python script for plotting it in a time-series graph.

Appendix B

Experiment A: Remaining Graphs

B.1 Concert Day 1 and Day 2: Time-Series Graph

The Figures B.1, B.2, B.3, B.4, B.5, B.6 illustrates the breathing samples collected during the concert on April 3rd of 2019—with a duration of one hour (between the time 19:00–20:00).

The Figures B.7, B.8, B.9, B.10, B.11, B.12 illustrates the breathing samples collected during the concert on April 4th of 2019—with a duration of 50 minutes (between the time 20:10–21:00).

The device spesification can be found in Table 6.1, and are mapped as: (A, B, C) Samsung Galaxy S9; (D) Google Pixel XL (version: 9.0); (E) Google Pixel XL (version: 7.1.2); and (F) OnePlus 3T.

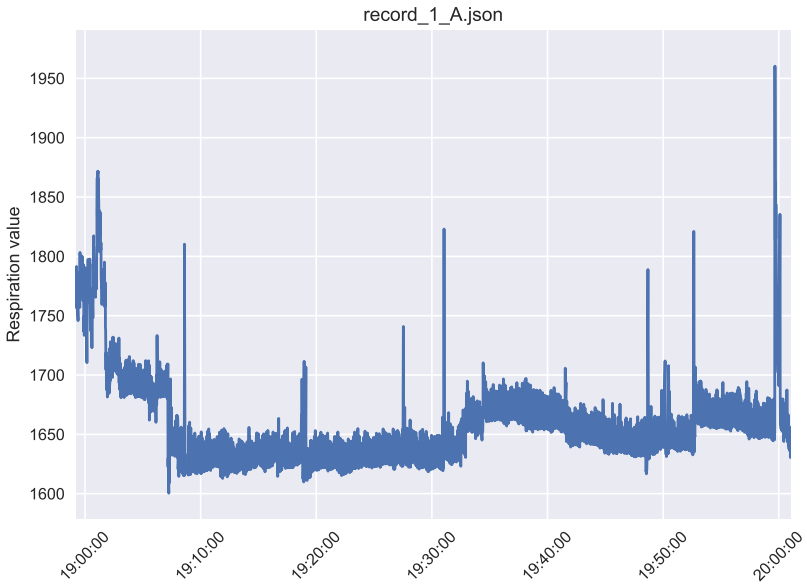

Figure B.1: Concert Day 1: Device Model A.

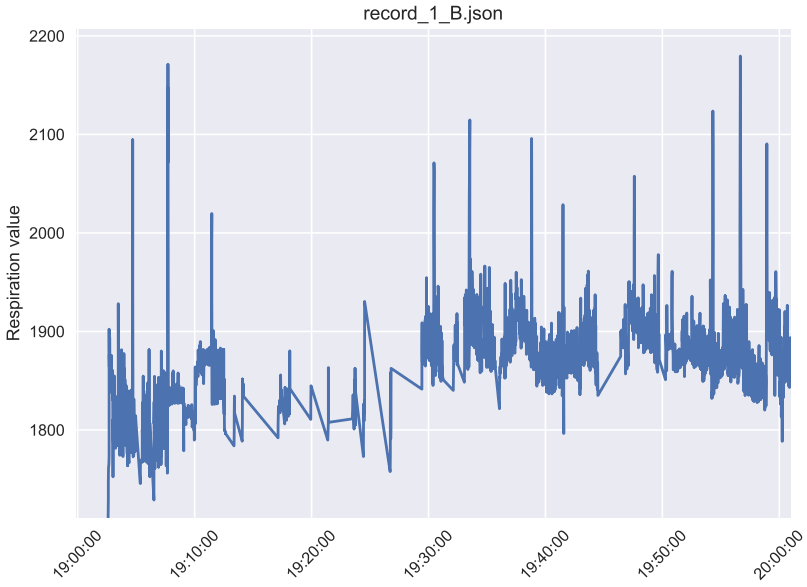

Figure B.2: Concert Day 1: Device Model B.

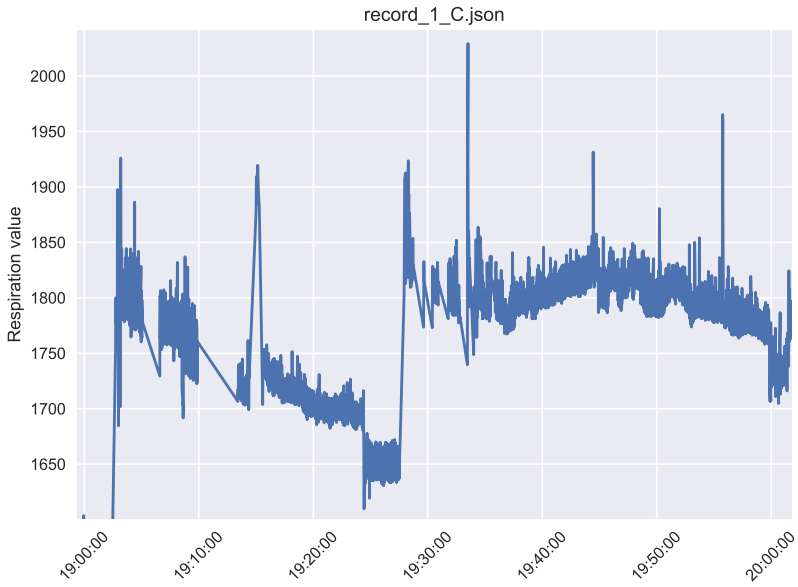

Figure B.3: Concert Day 1: Device Model C.

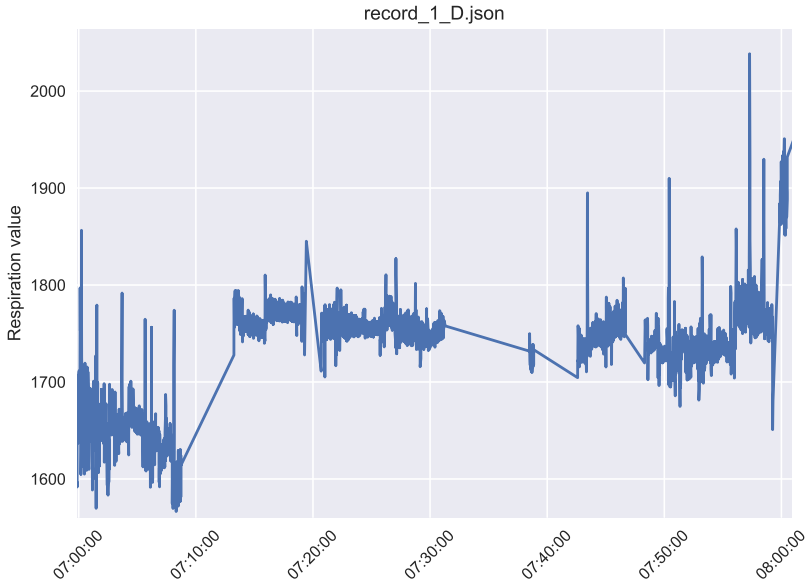

Figure B.4: Concert Day 1: Device Model D.

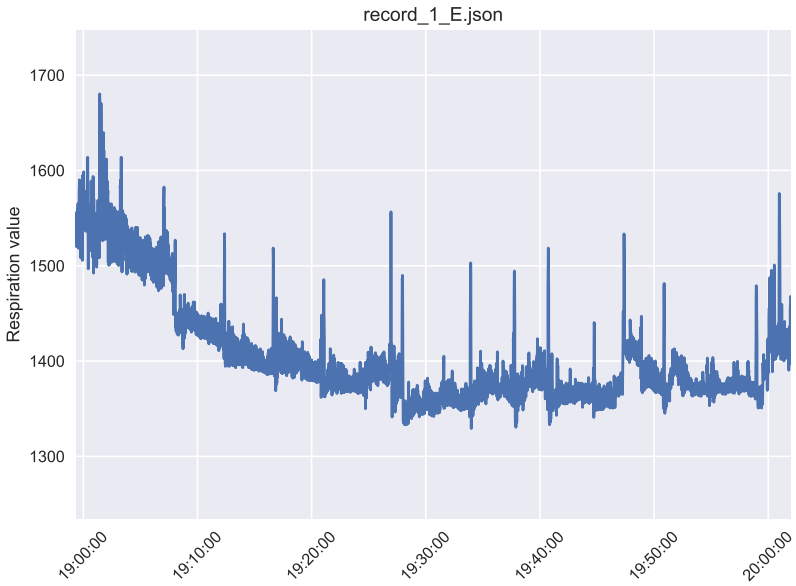

Figure B.5: Concert Day 1: Device Model E.

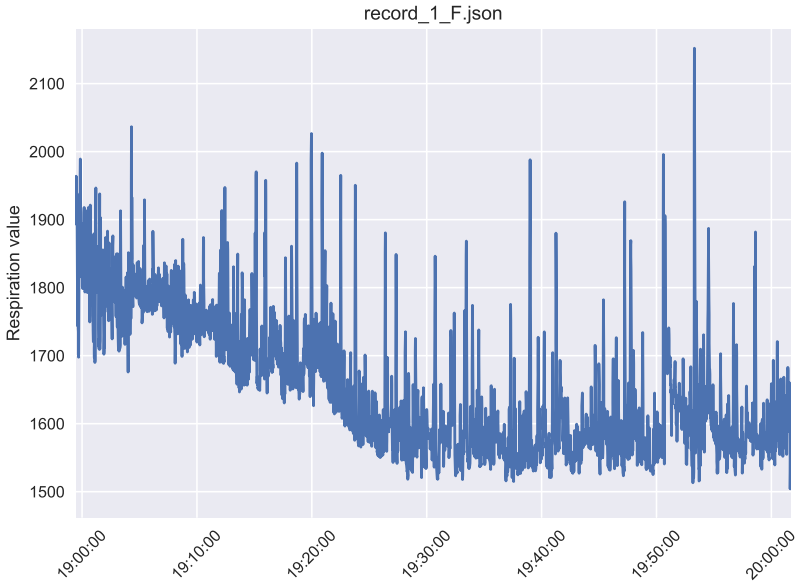

Figure B.6: Concert Day 1: Device Model F.

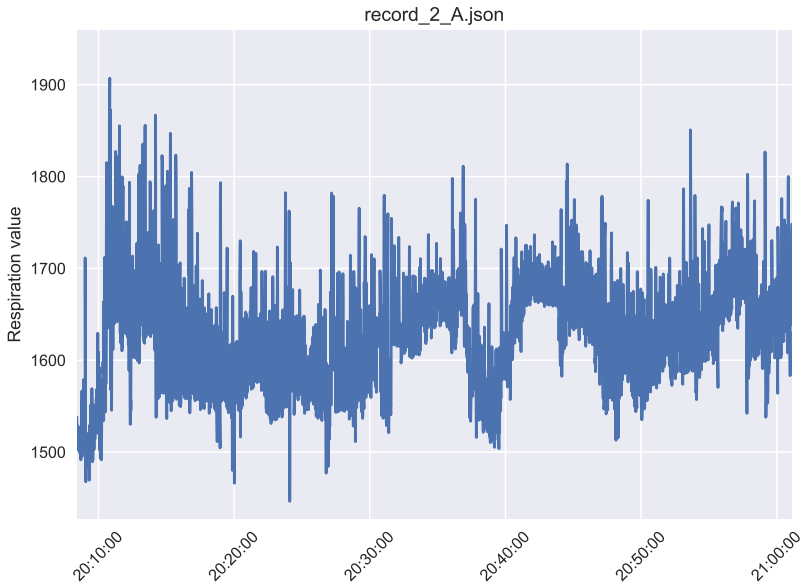

Figure B.7: Concert Day 2: Device Model A.

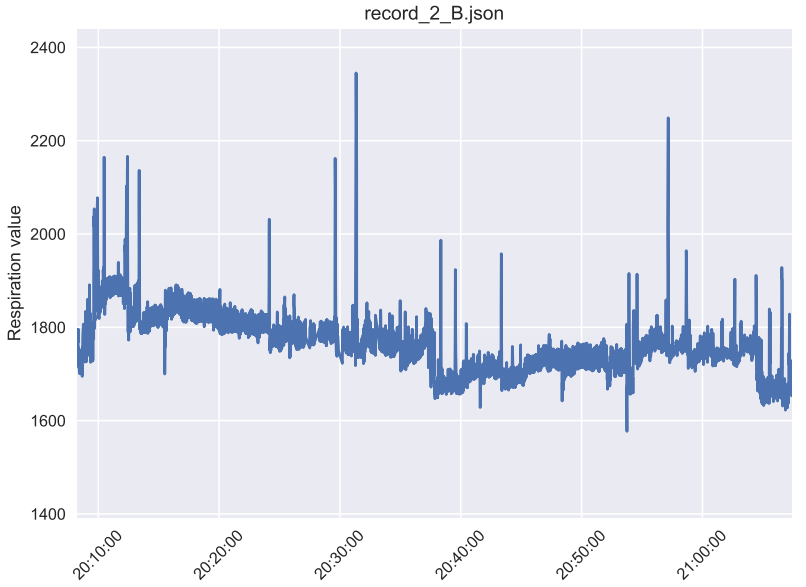

Figure B.8: Concert Day 2: Device Model B.

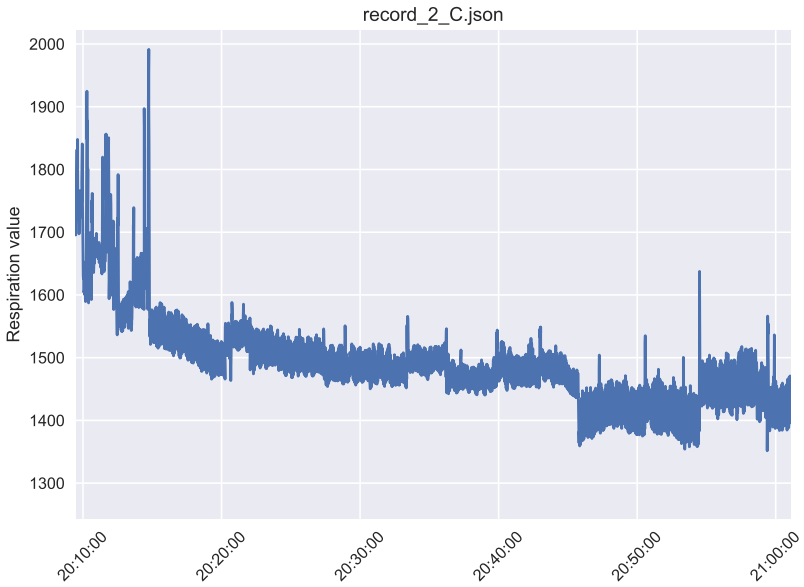

Figure B.9: Concert Day 2: Device Model C.

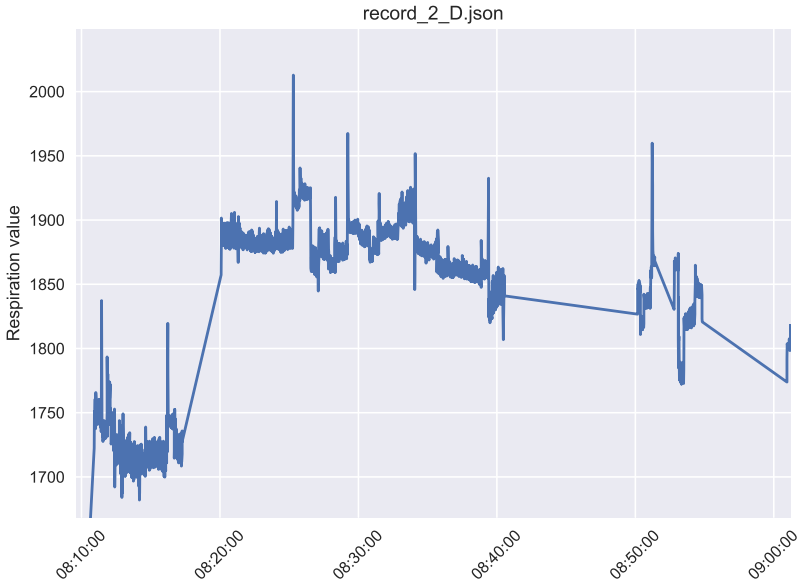

Figure B.10: Concert Day 2: Device Model D.

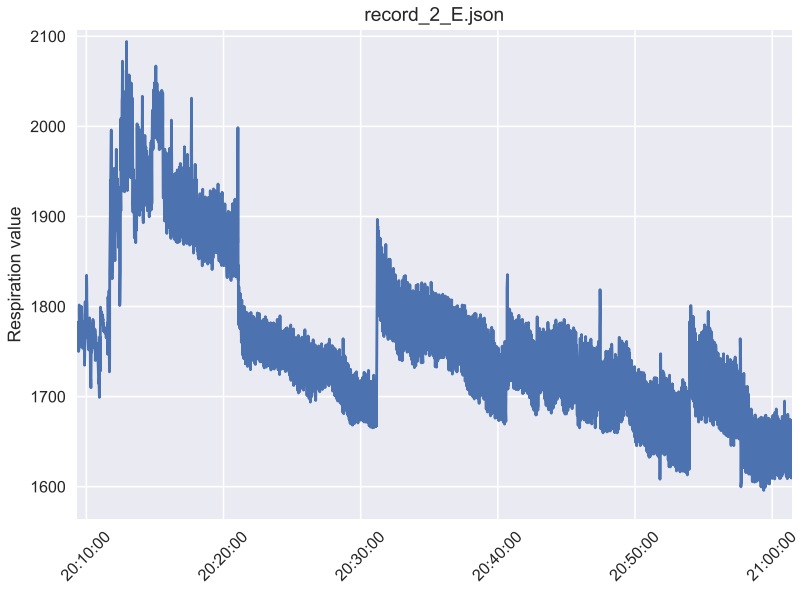

Figure B.11: Concert Day 2: Device Model E.

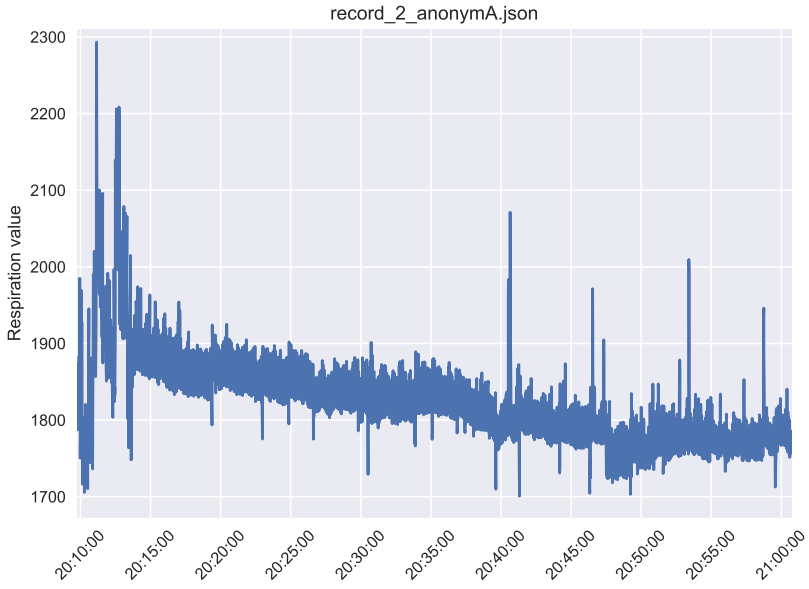

Figure B.12: Concert Day 2: Device Model F.

B.2 Python Code for Plotting

The following listing presents the code used for plotting the data from the concert into a time-series graph.

```
import json
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
from datetime import datetime
from statistics import mean
import sys
import seaborn

plt.style.use('seaborn')

sample_count = 0

def get_data(sample):
    data = sample[2].split("=")[1].split(",")
    avg = mean([int(i) for i in data])
    return avg

def get_date(sample):
    global sample_count

    if sample > '20:10:00' and sample < '21:00:00':
        sample_count += 1

    return datetime.strptime(sample, '%H:%M:%S')

def parse(data, name):
    global sample_count

    date = [get_date(i['implicitTS'].split(" ")[-1]) for i in
        data[0]['samples']]
    data = [get_data(i['sample'].split(", ")) for i in data[0]['
        samples']]

    fig, ax = plt.subplots()
    plt.plot(date, data)

    ax.xaxis.set_major_formatter(DateFormatter('%H:%M:%S'))
    ax.xaxis_date()
    ax.xaxis.set_tick_params(rotation=45)

    ax.set_xlabel("Time")
    ax.set_ylabel("Respiration value")
```

```
    ax.set_title(name)

    plt.show()

    print(sample_count)

def read(filename="record_1_B.json"):
    with open(filename) as f:
        json_data = json.load(f)

        parse(json_data, filename)

if __name__ == "__main__":
    if len(sys.argv) == 2:
        read(sys.argv[1])
    else:
        read()
```

Appendix C

Module Template

To expedite the creation of a new module, a template with pre-code is provided. The pre-code contains a blank Android project with the Gradle version 3.4.0 and support for Android 9 (API level 28). In the subsequent sections, the instructions of duplicating the template are presented.

C.1 Application Setup

C.1.1 Download the Application

Start by locating the *ModuleTemplate* application in the Github repository for this thesis. Download the application folder, rename it with the desired name, and move it to the `Android Studio` project folder. The listing below presents the commands to accommodate for these actions. Next, import the project in Android Studio by pressing `File --> Open --> (Name of Application)`

```
1 git clone https://github.com/Perelan/Master.git
2 cd Master
3 mv ModuleTemplate <Path to Android Studio Project>/<Desired
    Name>
```

C.1.2 Change the Name of the Application

Change the name of the application such that the user can locate the application in the app drawer; also, the name is used as the module-name when listed in Nidra. The name change is performed by locating the `app_name` inside of `strings.xml` which is to be found in `res/value`, see Listing below.

```
<resources>
    <string name="app_name">ModuleTemplate</string> <-- Change
        this!
</resources>
```

C.1.3 Rename the Package of the Application

Change the package name of the application by following the listing below. It is incentivized to keep the prefix of *com.cesar.X* to group Nidra-specific modules by the package name.

```
Right Click => Refactor => Rename => Rename Package => *new
    package name* => Refactor
```

C.1.4 Change the Application ID

Finally, change the application ID respectivly to the package name defined above—the ID is used to seperate installed applications from each other. The application ID is located in the `build.gradle` file inside of the `app` folder of the project. See the listing below.

```
android {
    defaultConfig {
        applicationId "com.cesar.moduletemplate"
    }
}
```

C.2 Application Execution

C.2.1 Add the Module to Nidra

In order to retrieve the records data, the module-application has to be added to Nidra. Therefore, launch the Nidra application on the device and navigate to the modules screen. Press the *Add new Module* button, find the module by name in the list and click to add.

C.2.2 Retrieve the Data

The data is sent as a JSON string to the module when it is launched within Nidra. The pre-code is separated into the following files and folder:

MainActivity On creation of the module-application, the bundle of data that is sent as an Intent is passed to the `Extract` method in `DataExtraction`.

DataExtraction The `Extract` method retrieves the JSON string from the bundle with the key *data*, and returns a unmarshalls of the data into a `PayloadFormat` object

Payload/PayloadFormat Is the object in which one of the record with corresponding samples of the data is marshalled. It encompasses a single record that contains metadata and user information, also a list of samples.

Payload/Record contains the record metadata (see Section 4.4.2)

Payload/Sample contains a single sample (see Section 4.4.2)

Payload/User contains the state of the user information at the given time of recording (see Section 4.4.2).

As part of the discussion of the module design, the records are sent to the module-application only when it is launched within Nidra. This might become cumbersome in the long run. Therefore, it is highly incentivized to temporarily cache the data under development.

Appendix D

Flow Sensor Wrapper

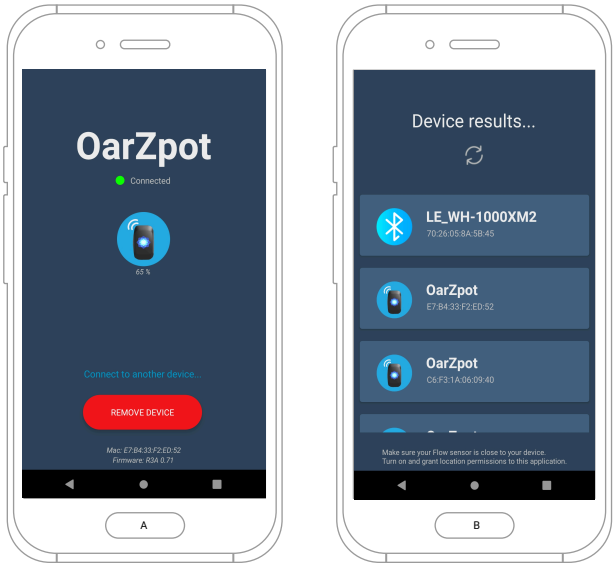

Figure D.1: The Flow sensor wrapper application presented to the user with following screens: (A) main screen, and (B) selecting a sensor source.

D.1 Implementation & Presentation

The Flow sensor wrapper was introduced in Section 5.1.1. The following sections describe the functionality of the sensor wrapper by separating them into following actions: (A) start the data acquisition; (B) handle the samples from the sensor; and (C) stop the data acquisition. The steps for the actions is presented in Figure D.2.

Moreover, Figure D.1 presents the Flow sensor wrapper application (called Flow), and the screens can be described as:

A The main screen that shows the state of the Flow sensor (e.g., battery level, MAC address, firmware version), including the button actions

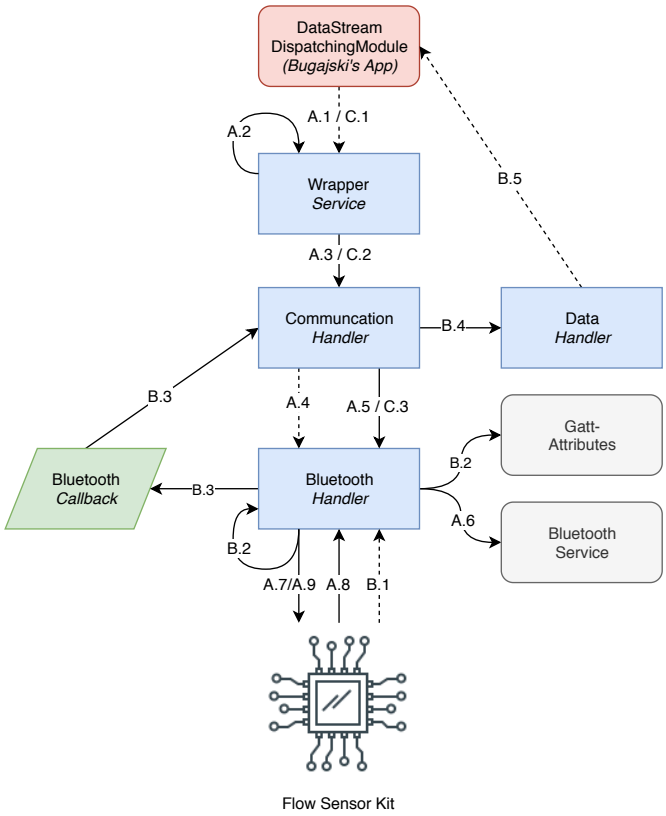

Figure D.2: Implementation of Flow sensor wrapper with the actions of: (A) start the data acacquisition, (B) handle the samples from the sensor; (C) stop the data acquisition.

to either remove the sensor or change to another Flow sensor.

B In the cases where there is no Flow sensor configured for the mobile device or the action to change the sensor, the device list screen display all available sensor or devices within range of BlueTooth connectivity to the user. The Flow sensors are distinctly from the other BlueTooth devices in order to make it easier to select.

D.1.1 Action A: Start the Data Acquistion

Action A is to start the data acquisition based on the command sent by the data streams dispatching module with a broadcast containing the *Start* message. The steps for this action can be narrowed down to:

A.1 A broadcast from the DSDM with an Intent of *Start* is sent to the `WrapperService`.

A.2 Within the `WrapperService`, a binder between the between this service component and the DSDM is created (which will be used for sending data by the Flow sensor). Moreover, a new thread for the `CommuncationHandler` is created and started.

A.3 This component initializes the connectivity with the `BluetoothHandler` service and listens for data packets from that component.

A.4 A reference to the `BlueToothHandler` with a binder is created, in order to communicate with the service.

A.5 Moreover, the request to connect and start the acquisition is sent to the `BluetoothHandler`.

A.6 `BluetootHandler` creates an object of the `BluetoothService`, which contains the state of the Flow sensor (e.g., battery level, breathing data-enabled, and manufacturer name).

A.7 The `BluetootHandler` connects with the GATT server (the Flow sensor), and once the connectivity is established (using the Android provided API's for connecting with the Bluetooth LE devices), we can send a request to the sensor source for available services.

A.8 The sensor source responds with all of the available services and characteristics, and we can proceed to notify the sensor source to enable the breathing data and the battery level service.

D.1.2 Action B: Handle the Samples from the Sensor

The Flow sensor sends data with a frequency of approximately 1.5Hz. As such, Action B is to handle the data and forward it to the data streams dispatching module. The steps for this action can be narrowed down to:

B.1 Periodically, the Flow sensor sends data to the connected mobile device, based on the activated service (i.e., breathing data and battery level). These data are sent to `BlueToothHandler`.

B.2 The `BluetoothHandler` handles the data received from the flow sensor (by filtering them out on GattAttributes) and reads the value from the characteristics that is encompassed in the service.

B.3 The values are sent as a callback to the `CommunicationHandler`.

B.4 Which further sends it to the `DataHandler`.

B.5 The `DataHandler` creates a JSON string that contains meta-data (e.g., the ID of the sensor wrapper) together with the data, and sends the data to the DSDM for further processing.

D.1.3 Action C: Stop the Data Acquestion

Action C is to stop the data acquisition based on the command sent by the data streams dispatching module with a broadcast containing the *Stop* message. The steps for this action can be narrowed down to:

C.1 A broadcast from the DSDM with an Intent of *Stop* is sent to the `WrapperService`. The `WrapperService` disconnect with the binder that was created between this component service and DSDM. As a result, the `CommunicationHandler` thread is interrupted.

C.2 The `CommunicationHandler` notifies the `BluetootHandler` to disconnect with the GATT server and terminate.

C.3 The `BluetootHandler` disconnects the connectivity with the Flow sensor wrapper, and further notifies the components that the communication is closed (e.g., the main screen changes the state from connected to disconnected on the main screen).