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

Acknowledgements

I would like to start of by thanking my supervisor, Professor Dr. Thomas Peter Plagemann, for his guidance throughout the work in this thesis. I am truely humble for the effort and dedication he put into helping me with this thesis. With my lacking experience of writing, this thesis would not be as complete. Therefore, I would sincere thank him.

Next, I would like to thank all of my friends that I gained during the study, as well as my childhood friends, for the encouragement. Above all, I show my gratitude towards my parents for their unconditional love and care, and to my Mom for without her encouragements and support I would not be where I am today. To her, this degree means much more than it does to me.

Finally, I would like to thank Sagar Sen at SweetZpot Inc. for giving me valuable insights on how to operate with the Flow sensor kit.

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

Mahatma Gandhi

Abstract

A vast majority of medical examinations requires the presence of a patient at the hospital or laboratory. Statistics Norway [33] 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. To overcome this hurdle, leveraging the technology that partakes in the diagnosis, treatment, and rehabilitation can make it more convenient for both parties. In recent years, the mobile phone has become significantly advanced and powerful devices. With the possibility for creating applications that users can interact with, and the support of built-in sensors and external sensors source for collecting physiological signals—such as breathing data and hearth rate data—can aid in examinations of specific illnesses or disorder from home. Applications that focus on improving healthcare are known as mHealth applications, and various applications in the real-world try to analyze, observe, and diagnose the health implications for human beings with mobile technology [3, 34, 35]. 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 improve the diagnosis of obstructive sleep apnea (OSA) [5]. The project facilitates a tool for managing and collecting physiological signals from external sensor sources (e.g., Bitalino) and the support for integrating future sensor sources.

In this thesis, we proceed to extend the project by designing and developing an Android application for patients to record, share, and analyze breathing data collected with the Flow sensor kit over an extended period. The motivation is to aid in collecting breathing data to potentially diagnose sleep apnea; albeit, the application can be used in other fields of study (e.g., physical activites). Also, we facilitate for an extensible application, that allows for future developers to create modules that extend the functionality in the application or enrich the data from the patient's records. The name of the application is Nidra—named after the Hindu goddess of sleep.

Contents

List of Tables

List of Figures

Listings

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Background and Motivation

The medical scenario of focus for this thesis is in in the field of sleep-related breathing disorders, which is characterized by abnormal respiration during sleep. Obstructive sleep apnea (OSA) is a category in disorder, which in layman's terms is when the natural breathing cycle is partially or complete affected 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 [cite]. As of now, the diagnosing of OSA is performed with polysomnography[1] in sleep laboratories. However, this method is both 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, resulting in an uncomfortable and inconvenient situation for the patient. 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 unit. As of now, mobile phones come with powerful processors, a sufficient amount of RAM, and adequate amount of battery capacity. Mobile phones operate with an operating system at its core; the operating system facilitates for 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, external vendors can create sensors that aid in detection of events or changes in an environment, and send the information to mobile devices. In the aspect of monitoring sleep-related breathing disorders, there are various sensor vendors which translate physiological

[1] Test used to diagnose sleep disorders

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. [cite]. that provides an affordable respiratory effort belt, which captures the respiratory effort using strain-gauges. The 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. Todays's health examinations are carried out by specialists with expensive medical refined equipment, which is stationary at hospitals and requires the patients physically presence. Mobile technologies[2] can overcome the hurdle of a patients presence, encourage behaviors to prevent or reduce health problems, and provide personalized/on-demand interventions [cite] in the hands of patients. As such, mobile technologies endorse healthcare solutions and opens up for possibilities within diagnosis and treatment of diseases outside the hospital. Levering mobile devices and external sensor vendors allows for innovation in mHealth application, which further allows for progress in the field of health research.

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 improve diagnosis of obstructive sleep apnea (OSA). The CESAR project focuses on *"development of new software solutions bridging state-of-the-art consumer electronic devices with appropriate sensors [...] to enable anyone to monitor physiological parameters that are relevant for OSA monitoring at home"* [cite]. This unfolds the potential to perform diagnosing of OSA from home with the aid of various sensors sources, as opposed to the current method of sleeping over at a laboratory. Also, it minimizes the costs of examinations and stress induced to the patients, as well as the hospitals. The CESAR project facilities for service to incorporate current and future sensor source, and managing the data acquisition from subscribed sensor sources. However, the project does not facilitate any application that exposes the functionality to record, share, or analyze the data acquisition to the end-user (i.e., patients or researchers/doctors).

To expand on the project, the motivation for this thesis is to utilize the tool for sensor management in order to create an application that provides an user-interface for the patient. The application should be able collect breathing data, share the recording across applications, and analyzing the data in a graph. Also, create an application that acts as a platform for other applications to leverage the recordings and perform advanced analytics or extended functionality in order to gather related applications into a generalized application.

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

1.2 Problem Statement

As indicated in the background and motivation section, we decided to look into opportunity of extending the the CESAR project even further. The market has new and affordable sensors that can aid with the data acquisition, which we seamlessly can integrate with the extensible data acquisition tool. The Flow sensor kit is a interesting sensor source due to its versatility and adaptability of collecting breathing data and connecting to devices with BlueTooth. As of now, the applications that support Flow sensor kit, are not user-friendly or designed to collect data over an extend period. Therefore, we look into designing and implementing an Android application that is used to record, share, and analyze breathing data over an extended period (e.g., during sleep). Also, we want to facilitate for an extensible application such that future developers can extend functionality of the application.

In the end, we can hopefully strengthen the detection of abnormal sleeping patterns, and decreasing the risk of the symptoms they may endure. Also, we want to create a generalized application that can be used in other fields of studies (e.g., physical activites). As the scope of the this thesis, we will be focusing on the completion of three main goals:

Goal 1 Integrate and support for Flow sensor kit with the *extensible data acqusition tool*

Goal 2 Research and develop a user-friendly application which facilitates collection physilogical data through the extensible data acquisition tool.

Goal 3 Create a "platform" solution for developers to create modules.

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 should provide an interface for the patient to 1) record physiological signals (e.g., during sleep); 2) present the results; and 3) share the results.

Requirement 2 The application should ensure a seamless and continuous data stream, uninterrupted from sensor disconnections and human disruptions.

Requirement 3 The application should provide an interface for the developers to create modules to enrich the data from records or extend the functionality of the application.

1.3 Limitations

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

We limited the scope of testing for existing sensor source development, and mainly focus on integrating the support for the Flow sensor kit in Nidra. Further, with the Flow sensor kit provided under development, we restricted the design to collect respiration (breathing) data (opposed to hearth-rate or other physiological data).

Additionally, the implementation is Android specific as the previous work performed on the project is designed soley for Android applications.

1.4 Research Method

The work in this thesis can be classified as *computing research* with a principle approach of an *engineering method* as described in [cite]. The engineering method (evolutionary paradigm) is to: (i) observe existing methods, (ii) propose better solution, (iii) build or develop artifacts[3], and (iv) measure and analyze until no further improvements are possible [CITE]. The report identifies patterns amongst various principle approaches and categorizes the the pattern 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"*

As part of this thesis is to design and implement an application that collects respiration data during sleep, and to future analyze and examinate the breathing data to detect sleep related-breathing disorders (e.g., Obstructive Sleep Apnea), we conducted a survey on previous related work. We found that a vast majority has designed and implementated mobile applications to

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"*

[3] human-manufactured objects produced during the development

Based on the related work, one of the goals of this thesis is to create an extensible application that allows future "related" applications to extend the functionality or enrich the data in our application. Also, to create an application which will be used to record, share and analyze respiration data collected over an extended period.

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"*

In the design chapther of this thesis, we analyze and explore various methods of implementing a functionality/task. In the implementation chapther of this thesis, we demonstrate the design choices by developing an Android application which follows the design.

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 performed various experiements to evaluate the performance of the application.

1.5 Contributions

Over the course of this thesis, we design and implemented an application for collecting, sharing and analyzing breathing data and a platform for modules to enrich the applications, called Nidra. The application is focused on creating a generalized application which manges sensor data with focus on sleep apnea / collecting data during sleep over an extended period, which purpose was to enable analyzing and sharing of the data with researchers/doctors.

Through the work produced in this thesis,

1.6 Thesis Outline

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

- Part 1: **Introduction & Background**

Chapther 2: Background presents the background material necessary for understanding the fundamentals in this thesis. It starts by introducing the CESAR project and the tools provided for data acquisition. Then, an overview of the Android operating system architecture and components are

Chapther 3: Related Work presents the related work focusing on creating a mobile applicaton to collect physiological data in order to diagnose sleep apnea. Finally, we discuss why our solution is an improvement to the related work.

- Part 2: **Design & Implementation**

  Chapther 4: Analysis and High-Level Design encompasses the functional requirements, the design purposal, and the structure of the data in the application.

  Chapther 5: Implementation realizes the design purposal in Android, with the use of the previous work.

- Part 3: **Evaluation & Conclusion**

  Chapther 6: Evaluation conducts various experiements in order to tests if the system requirements is fullfilling.

  Chapther 7: Conclusion discuss the objectives and the overall goal of the thesis, with suggestions to future work.

Chapter 2

Background

2.1 CESAR: Project Structure

The CESAR project is a collaberation between the Univeristy of Oslo and Oslo Universitetssykehus, with a goal to reduce the the threshold to perform a clinical diagnosis of Obstructive Sleep Apnea (OSA) and to reduce the time to diagonize the disorder. 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 [11].

The CESAR project aim to use low cost sensor kit to prototype applications using physiological signals related to heart rate, brain activity, oxygen level in blood to monitor sleep and breathing related illnesses (e.g., Obstructive Sleep Apnea). As of now, the development in the project facilities for data acqusistion from various sensor sources and delegation of the data to subscribed applications. The Section X and Section X is a summary of the development and results of the development, and Figure X is an illustration of the projects development pipeline.

Moreover, the previous work has incorporated support for a few sensor sources (i.e., Bitalino) in order to collect physiological signals. With the support for more sensor sources enables more precise detection and analysis of the disorder can be made. In Section X, a new sensor source is introduced.

The project has been developed by various people over the years, and the system has been divided into three parts (illustrated in Figure 2.1). The data acquisition part, the data streams dispatching part, and the application part. The first two parts are already implemented (summarized in the section below), thus, the last part is what we will be focusing throughout this thesis.

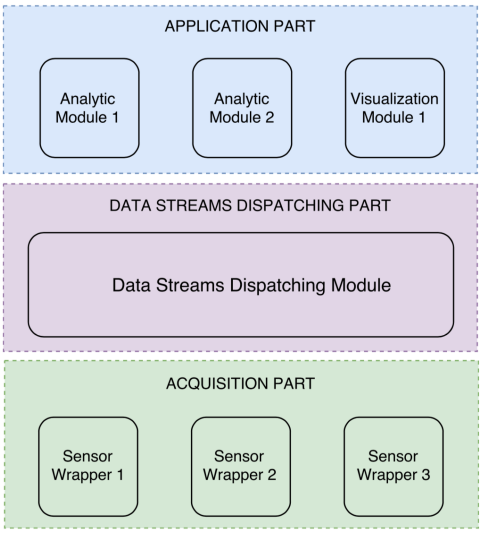

Figure 2.1: Structure of the project, separating functionality into three independent layers [10]

2.1.1 Extensible Data Acquisition Tool

In the thesis "Extensible data acquisition tool for Android" by Svein Petter Gjøby [23], we are proposed a *data acquisition system* for Android to make application development comprehensible. 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), whilst 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 address the following concerns regarding sensors:

- *Common abstraction/interface for the interchanged data*
  Sensor platform manufacturers have their own low-level protocol to support the functionality of their product. Typically, the manufacturers provide an software development kits (SDKs) to hide the low-level protocols so third-party development can be easier, however, both the low-level protocol and the SDKs are not standardized. Thus, for each sensor there might be exposure of 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, whilst 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, we gain the possibility to reuse the sensor wrapper application for different provider applications. However, there are some overheads with this solution. Mostly, the interprocess communication that might be costly and increase the complexity of the code. Nonetheless, the flexibility and extensibility gained by the 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 time stamp of when the data was sampled.

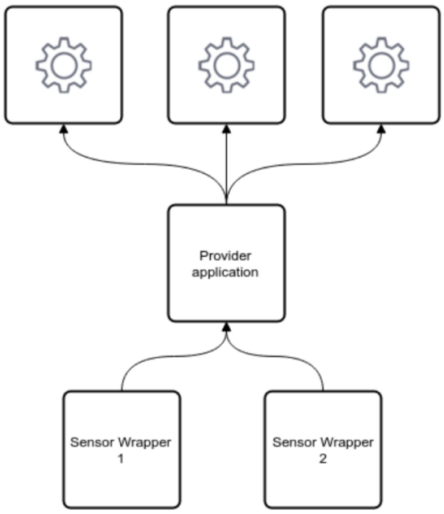

Figure 2.2: Sharing the collected data between multiple applications [23]

To summarize, the task of a *sensor wrapper* is to establish a connection to and collect data from exactly 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 visualized in Figure 2.2.

2.1.2 Extensible Data Stream Dispatching Tool

The extensible data acquisition tool developed by Gjøby leaves some space for improvements. Such improvements are discussed in the thesis "Extensible data streams dispatching tool for Android" by Daniel Bugajski [10]. Bugajski 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 extracting the functional requirement of the , and determining the responsibility of each element by. First, finding a model of all available data channels should be implemented. Then, developing a mechanism for cloning a data packet to allow reusing of data across modules. Finally, letting the modules 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 (i.e., from one channel), and duplicates the data several times - based on the number of subscribers.

3. *publish-subscribe* mechanism: is an interface responsible for providing possibilties of becoming a subscriber or a publisher, in addition, 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 should be collected with.

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.

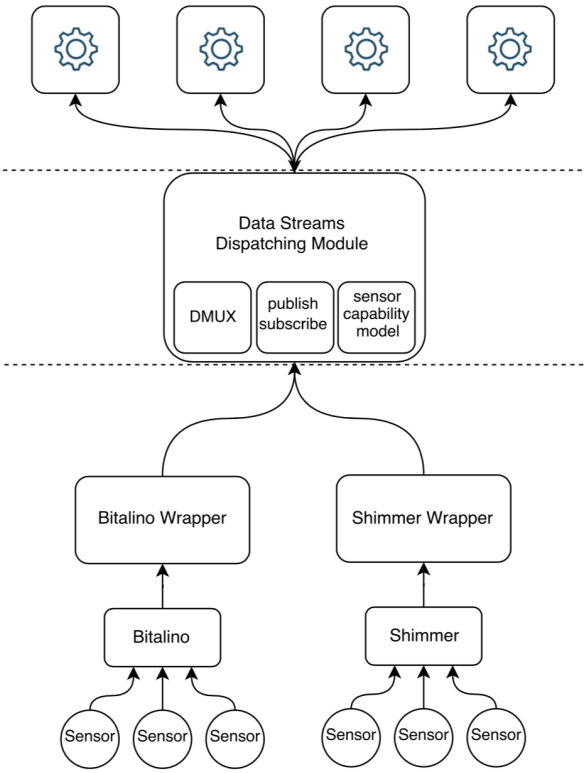

Figure 2.3: Sharing the collected data between multiple applications [10]

Addressing these issues leads us to the final architecture, which is presented in Figure 2.3, and meets the demands identified in the requirements. In this solution, all elements are placed between the application part and the wrapper layer, forming the data stream 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.

These three elements together establish *the data stream 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.

2.1.3 Flow Sensor Kit

Flow is an activity sensor created by SweetZpot, initially designed for measuring breathing and heart rate during activities. The sensor measures breathing per minute, in correlation to the heart rate, for an 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"* [41]. Thus, this sensor could be suiting for measuring sleeping problems in the project.

The sensor is a strap-on placed beneath the chest. It weights in at 27 grams, and a 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 an hand held device through Bluetooth technology, making it possible to connect with a mobile device.

The sensor is an affordable piece of equipment that we can utilize in the project in addition to the sensor we already have (i.e., BiTalino). The product has open-source library to support gathering sensor data, that we eligibility can use to analyze the collected data. Thus, creating a new wrapper for this sensor is required. Consequently, we can combine the sensor data with already integrated sensor in the system, to normalize the data and minimizing the variance for a more accurate measurement.

2.2 Android OS

Android is an operating system (OS) developed by Google Inc.

2.2.1 Android Architecture

The Android platform is an open-source and Linux-based software stack, containing six major components [36]:

- **Applications**: Android provides a core set of applications (e.g., SMS, Mail, and browser) pre-installed on the device. There are 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 a 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;

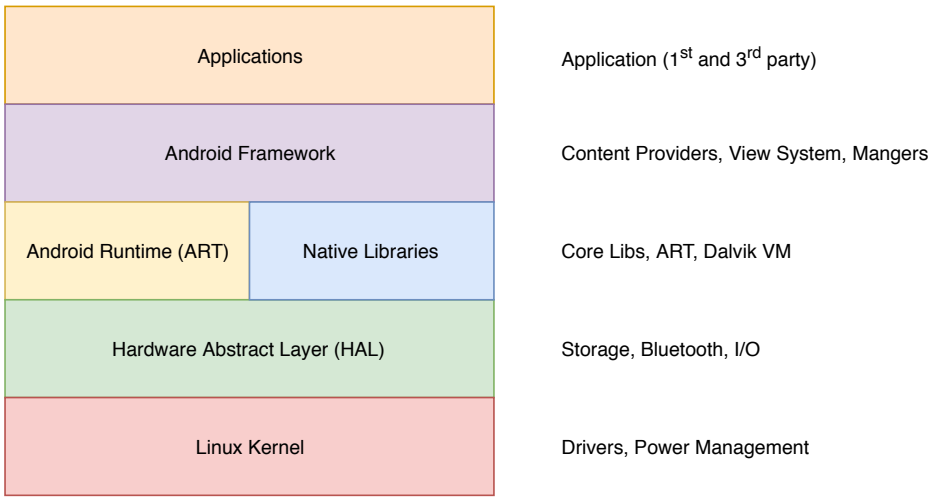

Figure 2.4: Recording

briefly characterized as *View System*: to build the user interface predefined 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 custome 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 its own process and has its own 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 specifcally 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 framwork. Hardware Abstract Layer (HAL) consists of multiple library modules that implements an interface for specific hardware components (e.g., camera, or bluetooth module).

- **Linux Kernel**: is the fundation of the Android platform. The ART relies on the functionality from the Linux kernal, 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 consists of four core components that are the building blocks of an Android application [7]. This section introduces these components; `Activites`, `Services`, `BroadcastReceivers`, and `Content Providers`. The activity is responsible for interactions with the user, services is a component that perform (long-running) tasks in the background, broadcast receivers handles broadcast messages from application components, and content providers manages shared set of application data. To enable the components, the Android system must be aware of the components existence. The existence of the components are defined in the manifest file (`AndroidManifest.xml`), which 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 conists of multiple acitvities, and an activity represents a single screen with a user interface [2]. Applications with multiple activities has to mark one of the activities as main activity, which will be presented to the user on launch. The user interface of an activtiy 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 activties remain in the stack (unless disgarded), and are brought back if desired. An activity can exist in three states cite[activites]:

- **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 acitvity does not cover the whole screen. A paused activity maintain its state, but can be killed by the system if the memory situtation is critical.
- **Stopped**: The activity is obscured by another activity. A stopped activity maintain its state; however, it is not visisible to the user and can be killed if the memory situtation 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.

Activites are part of a activity lifecycle, in Figure 2.5, the state of the activity can be vaugy 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 remaning 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 or effect the application occurs between these metods. As activties 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 are 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 [20]. Fragment allows for reuse of user interface or behavior across applications, and can be combinted to build a multi-pane user interface inside an activity. The fragment allows for more flexibility around the user interface, by allowing activties to comprise of multiple fragments which will have their own layout, events, and lifecycles. The lifecycle of a fragment is quite similar to the activiy lifecycle; with extended states for: fragment attachment/deattachment, fragment view creation/destruction, and host activity creation/destruction. A fragment is cooherent with its host activity, and the state of the fragment is affected by the state of the host activity. Fragment creation and interactions is done through `FragmentManager`.

2.2.2.2 Service

Service is a component that runs in the background to perform long-running tasks [40]. The application or other applications can start a service which remain in the background even if the user switches applications. In contrast, activites are not able to continue if 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

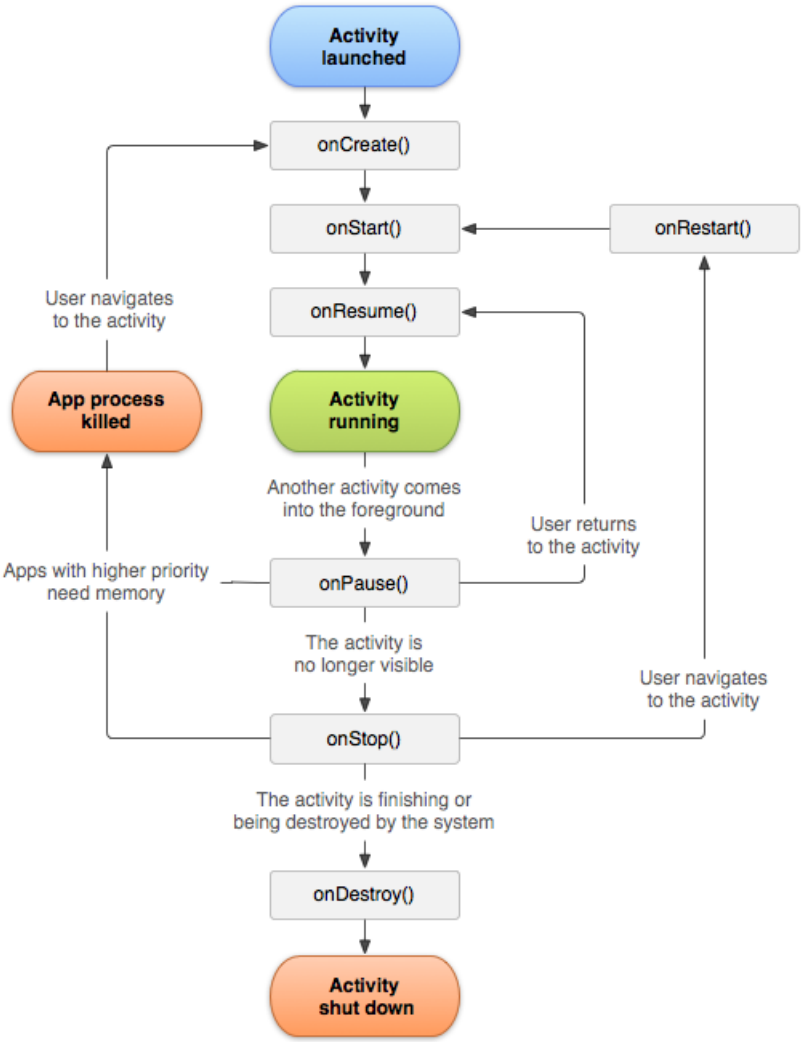

Figure 2.5: Recording

component 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., screen turned off, battery is low, or a picture was caputed). Applications can subscribe to messages, and the `BroadcastReceiver` can address and process the messages accodingly. 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 provides manage access to a set of structured data, and provide mechanism to encapsulate and secure the data [13]. 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 [19].

2.2.3 Process and Threads

The Android system creates a Linux process with a single thread of execution for an application on launch [38]. All components (i.e., activity, service, broadcast receiver, and providers) run in the same process and thread (called the *main* thread), unless the developer arrange for components to run in 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 consits of five levels (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 a often 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 controll (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 execute 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 activites. This process state has no impact on the user experiece, and these process are kept in a 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 activity is implemented correctly, to ensure a seamless user experience.

5. **Empty Process**: is a process that do not hold any active application components; however, are kept alive for caching and faster startup time for components that needs 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 serveral options: `runOnUiThread`, `postDelay`, and `AsyncTask` (perform asynchornous 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 back 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 transferrs them to the remote

process [8]. 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. Intent is a messaging object used to request action from another application components [25]. 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 genereal action to perform, which enables other applications to handle it. The main uses cases of an 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 [16]. 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 files saved in this directory is private to the application. Also, files stored in the interal 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 interal file storage. In addition, internal file storage allows for caching, which enables temprarily 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 a physicall removeable storage, such as SD card). Files stored into a external storage makes it possible to transfer files on a computer over USB. Files stored in a external file storage enables other applications to access the data, and the data remain available after application uninstallation (unless specified that the stoarge 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 uniquly identifying name, and the data is stored into a XML file. Also, the data stored remain persistent (even after application termination).

- **Databases**: Android provides support for SQlite databases, which is a relation database management system embedded into the system. The access to the database is private to the application, and accessing

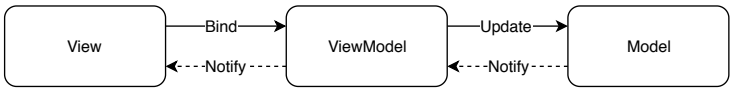

Figure 2.6: Entity Relationship Diagram

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 [24]. The Model-View-ViewModel (hereafter: MVVM) is an architectural pattern which is well-integrated and incentived by Android. It has three components that consititute 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 [30]. In Android, the `LiveData` is an obserable 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 [29]. Moreover, `Android Room` provides set of components to facilitate the structure of the model component [39]. More spesifically, 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 [37]. Android has feastures 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 network in the backgroud, and foreground services). The application that are restricted, functiontions 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 an user is not actively using it, resulting in the application background activity and jobs is 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 activites and tasks. The doze-mode enters maintancen window to complete pending work, and then resume sleep for a longer period of time. This cycles through until a maximum of sleep time is reached. Some applications wants to keep the device running to perform long-running tasks (e.g., collecting data), and `WakeLocks` enables this. `WakeLocks` allows application to perform activties and tasks, even while the screen is turned off [26].
- **Exemptions**: Another way of keeping an application awake, is to exemtping applications from being forced to `Doze` or to be in `App Standby`. The exmpted applications are listend in the settings of the device, and users can manualy choose application to exempt. Consequently, exempted applications might overconsume the battery of the device.

2.2.8 Bluetooth Low Energy

Android supports for Bluetooth Low Energy (BLE), which is desgned to provide lower power consumption on data transmittion (in contrast to classic Bluetooth) [9]. BLE allows Android applications to communicate with sensors or devices (e.g., heart rate sensor, and fitness devices), that has a stricter power requirements. Sensors that utilize BLE, are designed to last for a longer period of time (e.g., weeks or months before needing to charge or replace battery). The protocol of BLE is optimized for small burst of data exhange, 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 spesification for sending and receiving burst of data (known as *attributes*) over a BLE link.
- **Attribute Protocol (ATT)**: GATT utilites the Attribut Protocol, which uses a few bytes as possible to be uniquely identified by a Universall Unique Identifer (UUID). An UUID is a standardized format for identify information.
- **Characteristic**: Contains a single value and descriptors that describe the characterstics value (i.e., can be seen as a type).
- **Descriptor**: Are defined attributes that describe a characterstic value (e.g., specify a human-readable description, a range of acceptable values, or a unit of measure).

- **Service**: Is a collection of characterstics (e.g., a service is *Heart Rate Monitor* which includs a characterstic of *heart rate measurement*).

In order to enable BLE, factiliting for a GATT server and GATT client is required. Either the sensor or the device take 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 characterstics. And the GATT client can subscribe and read from the services the GATT server provides.

Chapter 3

Related Work

The widespread adoption of detecting and analyzing sleep-related breathing disorders on a mobile device has been a research topic and concert for some time. Various techniques and methods has been applied to detect sleep-related breathing disorders on a mobile phone with the use of built-in or external sensors. We will survey some of the research conducted in this field, and in the end, we will present similiarites based on our system.

Nandakumar, Gollakota, and Watson [34] presents 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 observe 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 clincal study with 37 patients, and concluded that the system managed to accurately compute the central, obstructive, and hyponea 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 smart phone. 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 tests. They achieved this by measuring the breathing patterns and movements patterns based on the built-in microphone and accelerometer in the smart phone. The system instructs the user to place the smart phone on its arm, abdomen or near the bed during recording. The data is collected on the smart phone, and sent to a centeral server in the cloud, where authroized docots 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 smart phone.

Penzel *et al.* [35] investigates the challenges and develop a system assoicated with insufficent conventional sleep laboratoris and their expensive

and time-consuming polysomnographic diagnosistics as early as in 1989. The purpose of the system is to make diagnose of sleep-related breathing disorders avaiable at any hosptial; 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 resolutaion of one value per second per parameter (e.g., mean value of respiratory frequency, ventilated volumn, actigraph actvity, and EOG actvity) and stored on a personal computer. After the recording, manual evalutation 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 basical evalutation and analysis.

3.1 Summary

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

To conclude this chapther, we can derive that the demand for a mobile system to detect sleep-related breathing disorders is high. Developing a system that records, analysis and detects sleep-related breathing disorders (e.g., Obstructive Sleep Apnea) enables users to diagnose the sleeping disorder from home, in contrast to a expensive and time-consuming polysomnographic diagnosistics at a labratory. Therefore, developing a system that is extensible and modular to new techiniques and methods, as well as for future sensors devices, is approriate in order to extend and evolve the research behind the detecting sleep-related breathing disorders on a mobile device. We developed an application for Android which enables recording, sharing and analytics, in addition to support for third-party modules to extend the functionality of the application, and to enrich the data collected on a patients device. The application is called *Nidra*, which we present in the next chapter.

Part II

Design and Implementation

Chapter 4

Analysis and High-Level Design

It is the goal of this thesis to enable detection of sleep-related illnesses with the aid of an Android device and low-cost sensors, and to further analyze and evaluate sleep- and breath-related patterns. We developed an application, called *Nidra*, which attempts to collect, analyze and share 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 motivation behind this application is to provide an interface for patients to potentially run a self-diagnostic from home, and to aid researchers and doctors with analysis of sleep- and breathing-related illnesses (e.g., Obstructive Sleep Apnea). An overview of the Nidra application pipeline can be found in Figure x, beginning with data acquired from a sensor, and ending with the data in the Nidra application. As for now, Nidra consists of three main functionalities, each related to the requirements defined in Section [Problem Statement].

1. The application should provide an interface for the patient to 1) record physiological signals (e.g., during sleep); 2) present the results; and 3) share the results.

2. The application should provide an interface for the developers to create modules to enrich the data from records or extend the functionality of the application.

3. The application should ensure a seamless and continuous data stream, uninterrupted from sensor disconnections and human disruptions.

This chapter will give a detailed look at the design of Nidra, including the tasks which constitute the structure of the application, the seperate concerns in relation to the tasks, and the structure of the data in the application.

4.1 Requirement Analysis

4.1.1 Stakeholders

McGrath and Whitty [32] describe the term stakeholder 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* has control of the project. In Nidra, there are three stakeholders who affect the application, and each can be categorized respectfully:

- `Patients`: Are identified as an end-user; they interact with 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 functionality in the application.
- `Developers`: Are identified as a contributor stakeholder; they maintain the application from bugs or extend the functionally of the application. 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 ROM, 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 [28]. The average battery capacity of a mobile device is approximately 2000 mAh on budget devices and around 3000 mAh on high-end devices [18]. 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 least amount of battery resources during a recording. Also, ensure sufficent 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

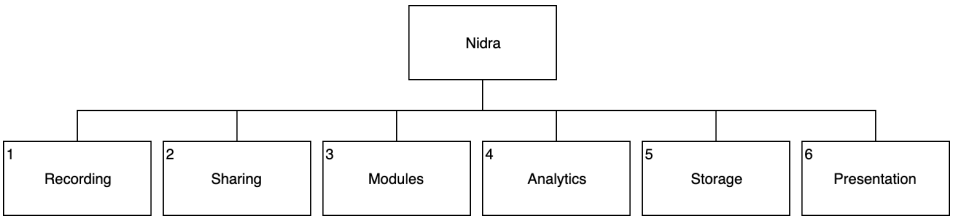

Figure 4.1: Recording

[43]. 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 [4] 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 [14]. In Figure 4.1, an illustration of the the tasks of the application is presented. This Subsection will introduce these tasks, which are an integral part to the development of the application.

Recording

A *recording* is a process of collecting and storing physiological signals 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 storing 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 over an application programming interface (API). A *sample* is a single sensor reading containing data and metadata, such as time and the physiological data. During a recording session, ensuring for a consistent and uninterrupted data stream from the sensors is vital to obtaining persistent and meaningful data. 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. [skrive mer?]

Module

A *module* is an independent application that can be 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 physiological signals, which enables the detection and analysis of sleep-related illnesses. There are various analytical methods, ranging from graphs to advanced machine learning algorithms. Incorporating a simple time series plot can indirectly aid in the analysis. For instance, plotting a time series graph where the physiological signals 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 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 layout and view of the application, 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 [27]. The concerns in this thesis are the individual tasks defined in task analysis. Each concern is conceptualized with a graph of components, the functionality of each component when combined constitutes a structure. The structure of each concern is derived based on research and development. In this Section, we will analyze and decompose the tasks defined in task analysis into subtasks (hereafter: components), where each component is functionality

4.3.1 Recording

The structure of a recording is restrictive in terms of arranging the components due to the design of CESAR. There are numerous ways of presenting the recording view; however, a recording structure is limited to the components of starting a recording, establishing sensor connection, monitoring of samples, and finalize sensors and recording. Additional components can be incorporated to aid a recording without causing disruption. For instance, the connectivity state component (see Section 4.3.1.1) provides extended functionality to the recording structure. In Figure 4.2, the illustration of a recording structure with the components and their dependencies are shown:

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

1.1.1 `Select Sensors`: From the sensor discovery, we can choose preferable sensors sources.

1.1.2 `All Sensors`: more Straightforward, we sample from all of the available sensors.

1.2 `Sensor Initialization`: Once we have a list of sensors sources, we need to establish and initialize a connection with the sensors. 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 through an API or IPC connection. All data exchange will occur over the established interface.

1.4 `Connection State`: Based on sensors establishments we can proceed to either start or stop a recording.

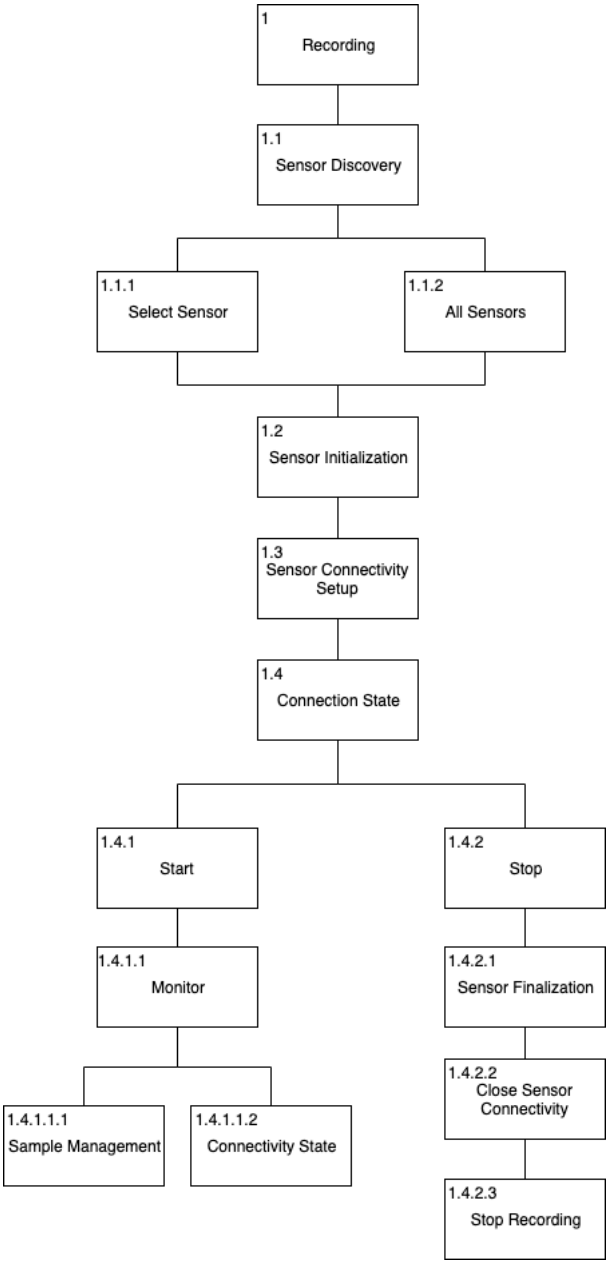

Figure 4.2: Recording

1.4.1 `Start`: By starting, we notify the sensors to begin collecting data, and the application should display that a recording has begun accordingly. 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 appropriately distributing 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 sensor (each sensor might have different payload structure), such that it is according to our data structure.

1.4.1.1.2 `Connectivity State`: Activly checking the state of sensor connectivty (read more below)

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 establishment between the application and the sensors.

1.4.2.3 `Stop Recording`: Once the sensors has closed its connections, 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 which will result in a lacking record. This component solves the issue of missing samples by actively reconnecting the sensor based on a time interval, resulting in more accurate record with fewer gaps between samples. 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 to 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 will be implementing the connectivity state in the recording structure.

2. A time interval triggers an event every specified time frame. If an event is triggered, a sample has not arrived, meaning the sensor either has been disconnected or disrupted. 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 is dead or completely out of reach can cause stress on the device. 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 separable 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 distributing 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, the structure of sharing is presented with components and their dependencies:

2.1 `Import`: Is a mechanism that locates a file, parse 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 Android document picker (ref) is more straightforward solution.

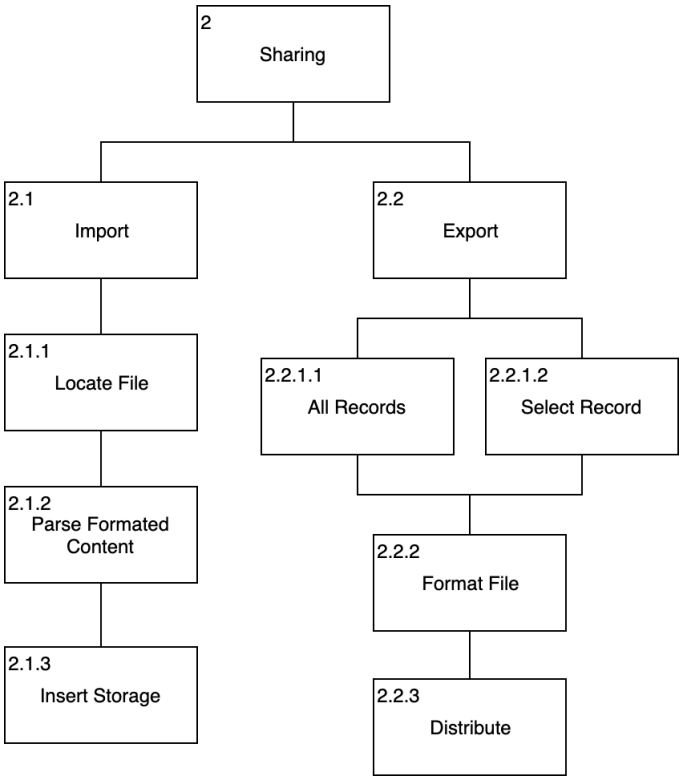

Figure 4.3: Sharing

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

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, format the record into a formatted (see Section 4.4.1) filed, and exports the file across device.

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 transmittal can be done. 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 across application (read more below).

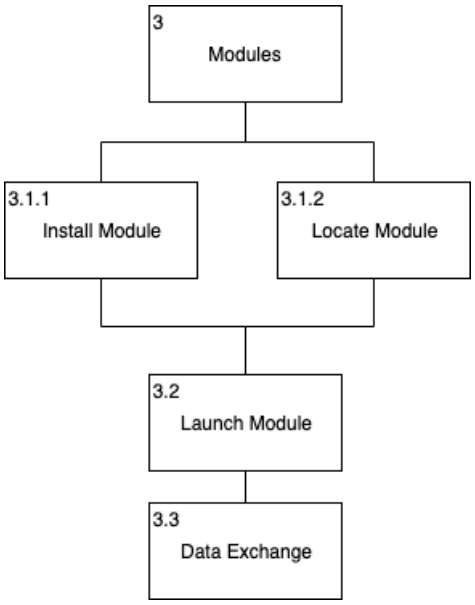

Figure 4.4: Modules

4.3.2.1 Distribute Component

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

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; however, 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 and data enrichment to Nidra. The components for locating and launching a module is limited to Android design; however, the component for data exchange between a module and Nidra can be designed variously. In Figure 4.4, the structure of modules is presented with components and their dependencies:

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 (read more below)

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 (Section Data) 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 module, 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-application, there are no ways of transmitting new data besides relaunching the module-application. 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, but that will require an interface for continuous data flow. For the simplicity of our design, we will be going with 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 illustration of the data in a time-series plot; 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`: locate the file on the device.

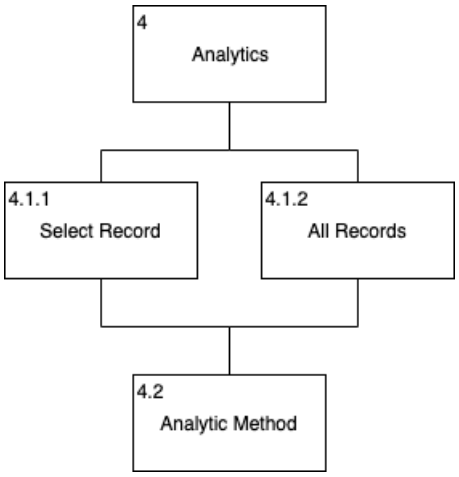

Figure 4.5: Analytics

4.1.2 `All Records`: parse the content of the file

4.2 `Analytic Method`:

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 [22]. 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 [21].

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.

4.3.5 Storage

Storage is the objective of achieving persistent data; data that is available after application termination. The data is characterized into four data

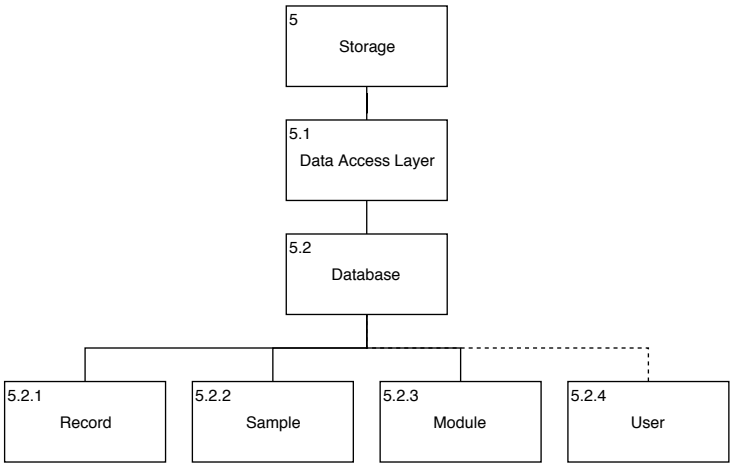

Figure 4.6: Storage

entities (i.e., record, sample, module, and user) that contains 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, the structure of storage is presented with components and their dependencies:

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

5.2 `Database`: Is the storage of all the data (read below).

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 is 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 is illustrated in Section 4.4.2.2.

5.2.3 `Module`: Is a table in the database, which contains fields appropriately to the sample structure. A sample contains the name of the module and a reference to the application package. The design decision and an example of a module record is 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 patients biometrical information (e.g., name, weight, height). The design decision and an example of a user is 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 [17]. Based on the options, some following design questions are 1) 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 (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 [31]. 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. 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 [31].

While a flat file database is applicable to store small and unchangeable data, it is not suitable for scalable and invasive data change. In Nidra, the samples acquisition makes it unreasonable to use a flat file database. Therefore, SQLite is a preferable solution to storing samples. Also, establishing a relationship between a record and a sample is made possible [rewrite].

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 derives from the functionality (concerns discussed) in the application, and research on the topic. In Figure 4.7, the structure of storage is presented with components and their dependencies:

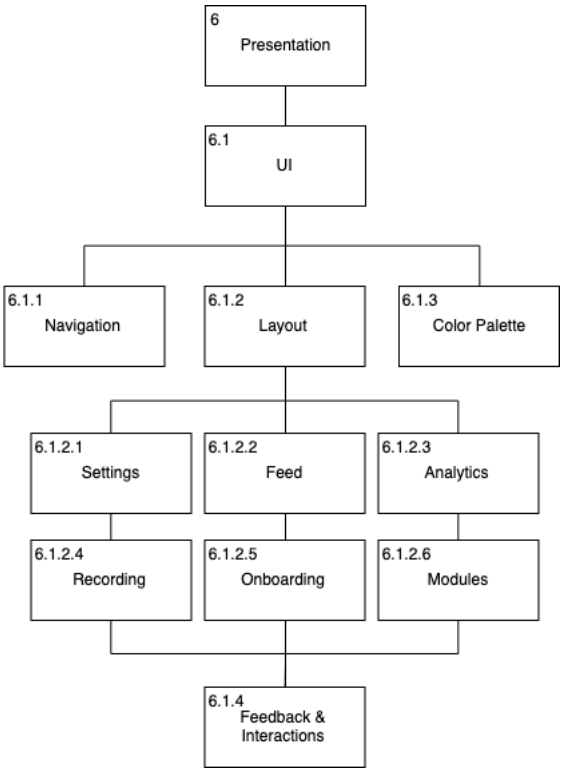

Figure 4.7: Presentation

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

6.1.1 `Layout`: Are the screen with the content of the current screen. The layout incorporates the color palette and some views have the navigation displayed.

6.1.2 `Navigation`: The navigation is a menu with options to change the layout.

6.1.3 `Color Palette`: A color selection that persist throughout the application (read more below).

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 layout has different feedback and interaction, which should be handled appropriately.

4.3.6.1 Color Palette Component

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 [42], 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 [15], 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. From this, we will be choosing a dark color theme.

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, transmittal 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. There are several data serialization formats; however, JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) are the two most common data serialization formats. In this Section, we will discuss these formats. In the end, we will 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, 44]. 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.

```
{
    "user": {
        "firstname": "Ola"
        "lastname": "Nordmann"
    }
}
```

Listing 4.1: My Caption

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 (see listing). [12, 44] XML provides mechanisms to define custom data types, using existing data types as a starting point, making it extensible for future data.

```
<user>
    <firstname>Ola</firstname>
    <lastname>Nordmann</lastname>
</user>
```

Listing 4.2: My Caption

4.4.1.3 Comparing

We will compare JSON and XML features and performance with the study conducted by Saurabh and D'Souza [44]. 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.

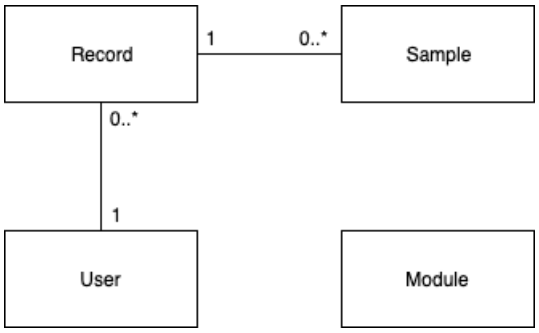

Figure 4.8: Modules

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 will be using 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. Subsection about Storage introduces four data entities in our application (i.e., user, record, sample, and module). In Figure 4.8, the relation between the data entities are shown. Record and sample stores information about the recording, and are separated into two individual entities in order to reduce data redundancy and improve data integrity. Although, samples have a reference to its record so they can be associated with each other. A user stores biometrical information related to the user, the user in the application are patients. A record 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 illnesses with relation to the biometrical information. A module 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 this Subsection, we will demonstrate the properties of each data entity in Nidra; storage structure of the entities, and illustration of the data structure for each entity.

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

Table 4.1: Example entry in record table

4.4.2.1 Record

A record is a table in the database that stores metadata related to the recording session. In Table 4.1, an illustration of the structure of a record is shown, with an entry of dummy data. In Nidra, the fields in the table for a record describe the data that is stored, which is separated into:

- `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 over 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`: Giving a rating on how the sleeping session felt, in a range between 0-5.
- `User`: User's biometriocal information encoded into a JSON string format, in order to capture the state of the user at 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).

4.4.2.2 Sample

A sample is a single sensor reading containing data and metadata related to the recording session. Samples are stored separated from a record; however, they are linked with a field in the table. In Table 4.1, an illustration of the structure of a sample is shown, with an entry of dummy data. In Nidra, the fields in the table for a sample describe the data that is stored accordingly to the data provided by Flow SweetZpot (ref), which are separated into:

- `ID`: Unique identification of a sample, also a primary key for the entry.
- `RecordID`: An identification to its correlated 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.

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 sample table

id	name	packageName
1	OSA Predicter	com.package.osa_predicter

Table 4.3: Example entry in module table

- `Sample`: Sensor reading contains metadata and data according to Flow Sweetzpot. The sensor aggregates seven samples in a single sensor reading.

4.4.2.3 Module

A module is a table in the database that stores all modules installed by the user in the application. In Table 4.3, an illustration of the structure of a module is shown, with an entry of dummy data. In Nidra, the fields in the table contain the name of the module and the reference to the module and can be summarized 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.

4.4.2.4 User

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 the context of detecting sleeping illnesses with the relation to the biometrical data. In Nidra, the user entity is not a part of the database, but as an object stored in the 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 Listing X, an illustration of the structure of a user is shown, with dummy data. In Nidra, the user object is structured in a JSON format and

```
{
    "user": {
        "name": "Ola Nordmann",
        "age": 50,
        "gender": "Male",
        "height": 180,
        "weight: 60
    }
}
```

Listing 4.3: My Caption

contains biometrical data related to the user:

- `Name`: Name of of the patient.
- `Age`: Age of the patient.
- `Gender`: Gender of the patient.
- `Weight`: Weight of the patient in kilograms.
- `Height`: Height of the patient in centimeters.

4.4.3 Data Packets

Data packets are parcels of data that Nidra receives from external applications (e.g., sensor wrappers) or send to other application (e.g., sharing). From the design choice In the Section above, the format of all the desired data should be according to the JSON format. In this Section.

4.4.3.1 Sharing

In Section 4.3.2, a proposal to the structure of exporting and importing data is discussed. Two of the components (`Parse Formatted Content` and `Format File`) uses JSON to either encode or decode the data. Listing 4.3 illustrate the content of the encoded data from our application to gain a broader understanding of how the data exchange in sharing operates. The attractive attributes from the encoding are the record (ref) and the samples. A record is an object that contains meta-data with name, number of samples, recording time, creation date, and user information. Samples are an array of objects that contains data, timestamp, and identification to correlated record.

```
{
    "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 4.4: My Caption

4.4.3.2 Sensor Data

Sensor data is data acquisition through the Data Dispatching (section background). The data format discussed in (section background).

Chapter 5

Implementation

5.1 Application Components

The CESAR project is introduced in the Background Chapther; to summarize, the data dispatching module discovers and connects with sensor wrappers (each sensor source has its own senor wrapper), and enables data acquesition to applications that subscribe to the data. As for this thesis, we operate with three different applications: Nidra, Data Stream Dispatching Module, and the sensor wrapper for Flow sensor kit.

Figure 5.1 illustrates the Android components (i.e., activity, service and broadcast receivers) for each applications. All of the applications run in a separate process on a device. In order to perform remote procedure calls (RPC) to application components that run remotely, we can use the IPC mechanism. In Android there are two mechanisms to mechanisms to enable IPC: (1) `Binder` enables a process to remotely invoke functions in another process; and (2) `Intent` a message passing interface aollowing applications to send messages to each other.

In the subsequent subsections an brief overview of the structure of the applications are discussed.

5.1.1 Data Stream Dispatching Module

The Data Stream Dispatching Module developed by Bugajski provides an interface for application instances to subscribe for data packets from connected sensor sources. The modularity this module provides towards managing and supporting various sensor capabilities, this allows for a faster development time (?). To briefly introduce the steps that enable this component, we will list the steps the component performs when there are one availble sensor source and one application instance subscribing for data for the named sensor source:

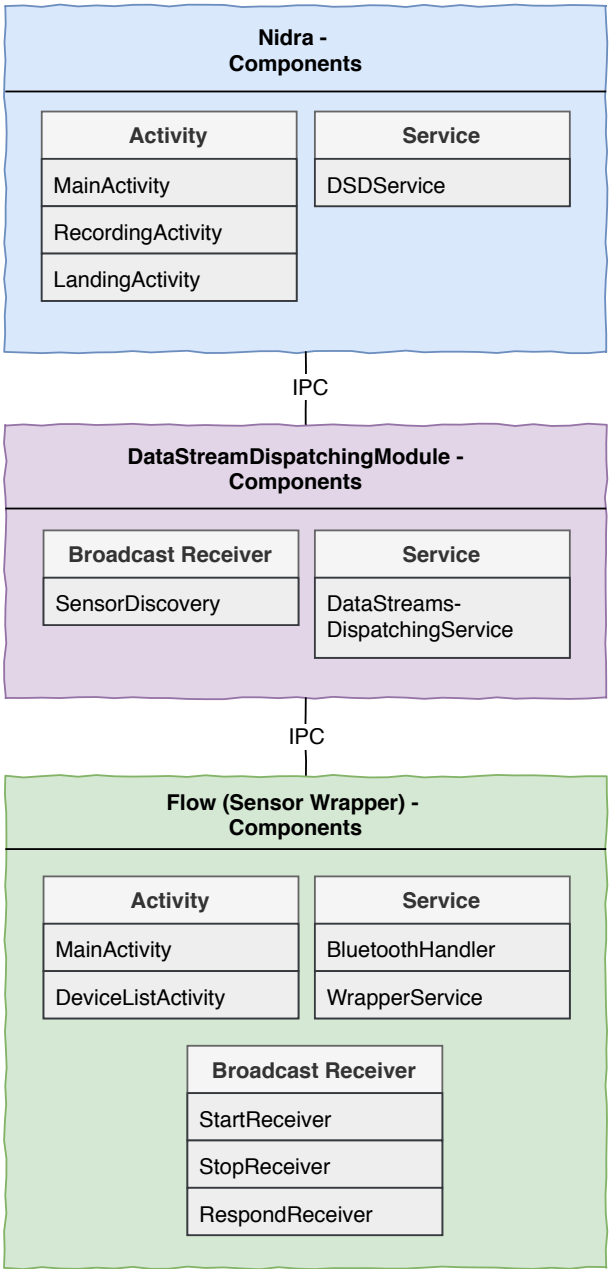

Figure 5.1: Applications components

Sensor Discovery The inital design of discovery for new sensor wrappers was performed as following: (1) the DSDM sends out an broadcast and with an action of *HELLO* in the Intent to discover all available sensor wrappers on the mobile device. All sensor wrappers are designed to listen for this event, and respond back to DSDM with their packagename as *id* and the name of the sensor wrapper as *name* with a broadcast to *REGISTER*. The DSDM is then aware of which sensor wrappers that are available on the mobile device.

However, during the development of this thesis, Android had limited and strictend the use of implicit broadcasts on newer Android versions [cite]. Implicit broadcasts are those broadcast that do not target a specific application, however, sends out an action with a message and those application that filters and listens for the actions can proceed to perform their actions towards this message appropiately. To overcome this problem, a re-design of the sensor discovery were made. Instead of DSDM ever so often sends out a HELLO broadcast, the sensor wrapper sends out an explicit broadcast directed to DSDM make it aware of its existance. The broadcast is sent to the *SensorDiscovery* directed explicitly to DSDM broadcast receivers, encapsulated with the name and the packagename of the sensor wrapper. The DSDM stores the sensor wrapper information in a `SharedPreference`. Upon launch of the DSDM, an indentical...

5.1.1.1 Start and Stop

5.1.2 Flow Sensor Wrapper

As part of the thesis objective is to integrate the support for the Flow sensor kit, we developed a sensor wrapper to establish a connection with the sensor capabilites with the DSDM. We followed the instructions to create a new driver application created by Gjøby, and the brief overview of main the component in the template are:

WrapperService Is instansiated by the DSDM during the sensor discovery phase. The events to handle start and stop of the data acquesition is managed within this service, as well as creating an IPC connection between modules (?).

CommunicationHandler When the data collection is signaled to start by the DSDM, a seperate thread in `CommunicationHandler` is created for communication with the sensor source. The connection is persistent and sends all of the data from the sensor source to DSDM until the signaled to stop (by the call of `interrupt()`). The connection, collection and disconnection assosicated with the data source is implemented in this thread.

DataHandler Is responsible for preprocessing the collected data, before sending to the DSDM application. Part of this process is to construct

the data packet correctly. The packet contains the `id` of the sensor wrapper, the current date and time, and the data with the samples. The packet is then sent using the establish binder connection, in order to send the data packet (on `putJson()`).

Besides the components which manages the connectivity, collection and disconnection with the sensor source, there are two activites that are resposible for selecting the sensor source, and displaying the state of the sensor source on the users screen:

MainActivity Presents the state and informatuon of the connected sensor source. Currently, it presents the connectivty state (connected or disconnected), the battery level of the sensor, the mac address and the firmware to the connected Flow sensor, and the option to remove or connect to another sensor device.

DeviceListActivity All of the available devices close to the mobile device which have BlueTooth activated is listed for the user to pick. Devices that are assosicated with the Flow sensor kit (e.g., OarZpot) has an distingushable icon to make it easier to select the correct device.

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

The preciding components is a part of a template to connect with the data stream 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 protocols, which described in the background chapther, is designed to provide lower power consumption on data transmittion and sensors that utilize BLE is designed to last for a longer period. Below, an brief overview of how to establish a connection and interpret the collected data with the Flow sensor kit through BLE in Android. However, an intricate detail of implementation can be found in Appendix D.

5.1.2.1 Communication with the Flow Sensor Kit

The Flow sensor kit provides no SDK or API to manage the connection with the sensors. In order to manage the connection with the Flow sensor source, we have to use BlueTooth Low Energy (BLE) protocols, which is used for sensors that has a resitriction on battery capacity to communicate with devices. In the limitations, we decided to only focus on collection respiration (breathing) data. Thus, we are

To begin with, the user has to select desired Flow sensor kit to use for collecting the data. As such, when the command of starting the collection is passed to the `StartReceiver` broadcast receiver in the application, an seperate 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 connection to the sensor source, discovers services provided by the sensor source, and manages decoding of the data received from the sensor. This component, acts as an GATT client which connects with a GATT server. The GATT server in our case is the Flow sensor kit, which provides a services that encompasses several *characterstics* whichs contains values and descriptors. In Bluetooth, the objects are identified by an universally unique identified (UUID)[1], and there is a collection of assigned numbers to standard objects [CITE]. 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 categorize an indiviual characterstic. The most interesting characterstics to us, are the manifacturar name (prefix: 0x2A29), firmware revision (prefix: 0x2A26), battery level (prefix:0x2A19) and flow (breathing) measures (prefix: 0xFFB3). The latter characterstic's prefix is not a part of the standard, however, manifacturer defined prefix. Also, the latter characterstic

In order to receive flow data and the battery level from the sensor source, we have to enable it by notifying the GATT server. This can be performed by spesificying the service and the characterstics for the desired service and the underlying characteristics we want the values from. For example, to enable flow (breathing) data, we specify the service (prefix: OxFFB0) and flow measure (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 kit gatheres data a frequency of 10 Hz, however, the data from the sensor source is sent to the connected devices on approximatly 1.5 Hz. Which means, each packet of the received contains 5-7 data points with a timestamp of acquesition. We proceed to smooth out the data by averaging the values for a timestamp, which in statistically measures is to filter out misfits of values and finding estimate of value on a given time. The is then packed and sent to the `CommunicationHandler` which further sends it to the `DataHandler`. The `DataHandler` formats the data into a JSON string, and sends the data on the binder between the sensor wrapper and DSDM (created in `WrapperService`) in the method `PutJson()`.

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

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

5.1.3 Nidra

5.1.4 Inter-Process Communication: AIDL

To perform IPC using Android Interface Definition Language (AIDL) [cite] we need to define a programming interface that both the client and the service agree upon. In order to communicate with processes, the data objects has to be decomposed into primitives that the operating system can understand, and marshall the objects across the boundary. The AIDL interface is defined in an `.AIDL` file, and located in the `src/` directory of the hosting service application (DSDM), and other applications that binds to the service (Nidra and sensor wrappers). It is important 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 the hosting service application exposes (DSDM). In Nidra, some of the functionality is utilized to enable recording. More spesifically, `getPubishers()` method is used to get all of the sensors publishers (e.g., Bitalino provides multiple sensor capabilites ...), the `Subscribe(...)` and `Unsubscribe(...)` is used in order to subscribe and unsubscribe to a spesific sensor capability, and listing for events on the `putJson(...)`, which is used by the sensor wrappers to send data collected to DSDM, and further sent to all subscribing applications (i.e., Nidra).

```
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: My Caption

The connection is instantiated

5.2 Implementation of Concerns

In Section (ref) we conceptualized the tasks, by decomposing the tasks into components and discussing various techniques and design decisions

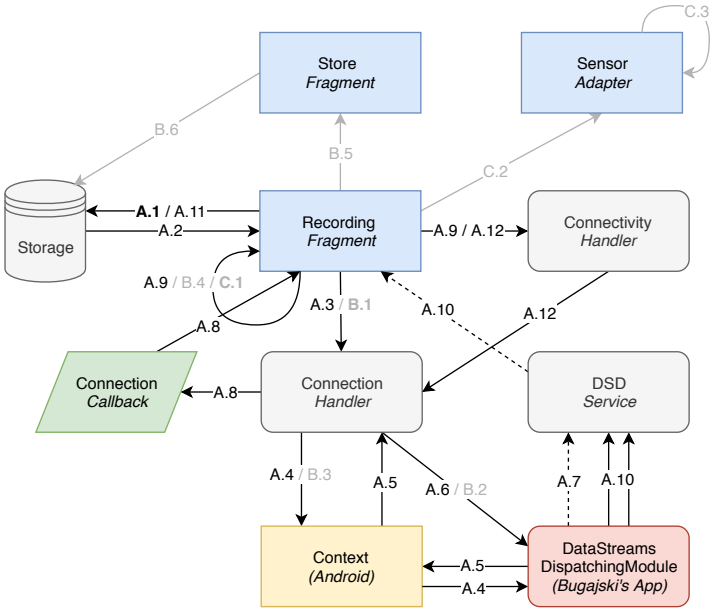

Figure 5.2: Implementation of recording functionality: (A) Start a Recording

for implementation. In this Section, we will realize the discussion by implementing the tasks in Android.

5.2.1 Recording

Recording is the process of collecting and storing physiological signals 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. We will use the Data Stream Dispatching Module (hereafter: DSDM) which manages sensor discovery and sensor establishment to supported sensor sources. DSDM facilitates an interface for data acquistion, and the communication between the DSDM and Nidra occurs over IPC. Moreover, we will implement functionality to check for connectivity during record, to ensure the sensors are collecting data to at appropriate rate. At the end of recording, we will store metadata related to the recording session and finialize the recording process.

The functionality of recording can be divided into three actions: (A) start a recording; (B) stop a recording; and (C) display recording analytics. In the following Subsection for recording, we will review the steps to enable these actions.

5.2.1.1 Action A: Start a Recording

In Figure 5.2, an illustration of the component interactions are shown. Action A is to start a recording by connecting and starting data aqusition

with the use of DSDM, and to ensure persistent connectivity with the sensor sources. The steps and interactions for this action are:

A.1 Commence the recording by creating a record entity and inserting it into the database. An empty record has to be inserted into the database in order to associate new samples to the recording.

A.2 Once the record is inserted into the storage, unique identification is returned.

A.3 `ConnectionHandler` manages the establishment, connection, and disconnection of the IPC between Nidra and DSDM. It starts by establishing a connection to the DSDM:

```
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);
```

Listing 5.2: My Caption

`MainServiceConnection` is the AIDL file (discussed in IPC).

A.4 We bind to service by using the `BindService`. If the service is offline, the flag `Service.BIND_AUTO_CREATE` will ensure the service is started. `BindService` allows components to send requests, receive responses, and perform inter-process communication (IPC) [CITE].

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

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. In the `Subscribe` function, a reference to the package name and a service object (`DSDService`) from Nidra is sent. The service object is where all of the parcels of data from the DSDM is received.

A.8 A callback to `RecordingFragment` with the available sensor publishers is made.

A.9 The recording has now started, and a timer to measure the time spent on the record is started.

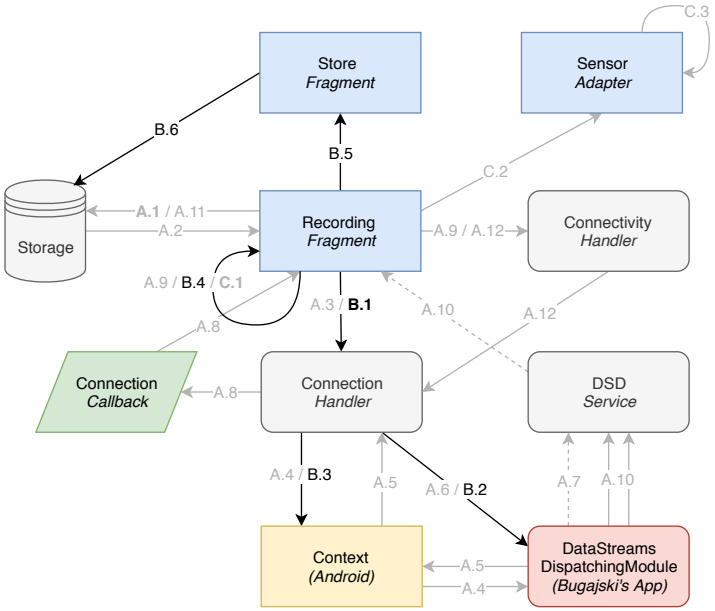

Figure 5.3: Implementation of recording functionality: (B) Stop a Recording

A.10 The `ConnectivityHandler` is the component we discussed in Section ... , which checks for sample arrival and activity to reconnect with the sensor. The `ConnectivityHandler` is implemented with a `Handler` with a `PostDelay`. If the event for the `PostDelay` is triggered, it is equivalent for a sample not being acquired from the sensor.

A.11 Periodically, the DSDM receives samples from the subscribed sensor. DSDM forwards the sample from the sensor to the service object (`DSDService) on the \verbputJson|` function. The DSDService uses a `LocalBroadcastManger` to send bundles of data across components in the application. The `RecordingFragment` is listening for the event, and receives all of the data.

A.12 The data received from the sensor (through the DSDM, to DSDService, and LocalBroadcast), is stored as a new sample entry with the current recording identification associated with it.

5.2.1.2 Action B: Stop a Recording

In Figure 5.3, an illustration of the components to stop a recording is shown. The action is based on the user input to stop the recording session. For the user, the recording has terminated and the user is presented with a screen to provide information regarding the recording (e.g., title, description and rating). For the application, has to notify and unsubscribe to the connected sensors through DSDM, as well as disconnect from the DSDM service. In addition, manage the metadata the user provides at the end of the recording session, and storing recording metadata on the device. The steps and interaction for this action are:

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

B.2 A call to the `Unsubscribe` function on the subscribed subscribe with the service object is sent to the DataStreamsDispatchingModule (hereafter: DSDM). The DSDM has to ensure to unsubscribe unpublish the sensor in order to signal the sensor to stop sampling.

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 enrich 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 and storing it in the record. 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 `FeedFragment`.

5.2.1.3 Action C: Display Recording Analytics

During a recording, the user can view the analytics for the recording. More spesifically, the user can see the available connected sensors, and a graph of respiration (based on the samples acquesitions from the sensor) in real-time. In Figure 5.4, an illustration of the components display recording analytics is 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 Graph library, we can in similarities to the implementation of the analytics concern (see Subsection), implement a graph to illustrate the respiration data to the user.

C.2 In addition to the time-series graph, we have a list of sensors that are sampling in the recording.

C.3 The `Sensor Adapter` populates and illustrates the connected sensor to the user.

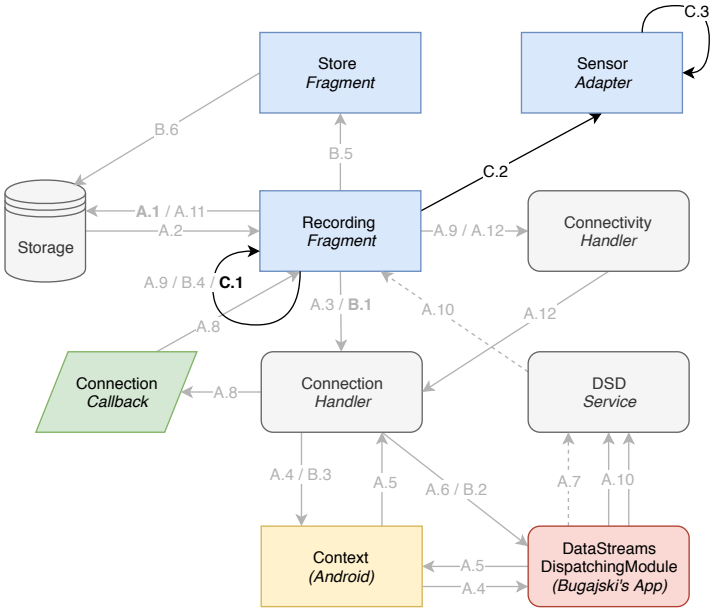

Figure 5.4: Implementation of recording functionality (C)

5.2.2 Sharing

Sharing enables users to transmitt records across application over a media. The functionality of sharing is separated into two concerns, namely importing and exporting records. Hence, the actions for sharing are separated into: (A) exporting one or all records; and (B) import a record from the device.

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

In this Subsection, we will take a look into the steps that are taken to enable the actions:

5.2.2.1 Action A: Exporting one or all Records

In Figure 5.5, 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 formated file, and prompting the user with options to select a media (e.g., mail) for transmittion. The steps can be narrowed down to:

A.1 Upon an event for exporting a selected record in `Feed Adapter`, the

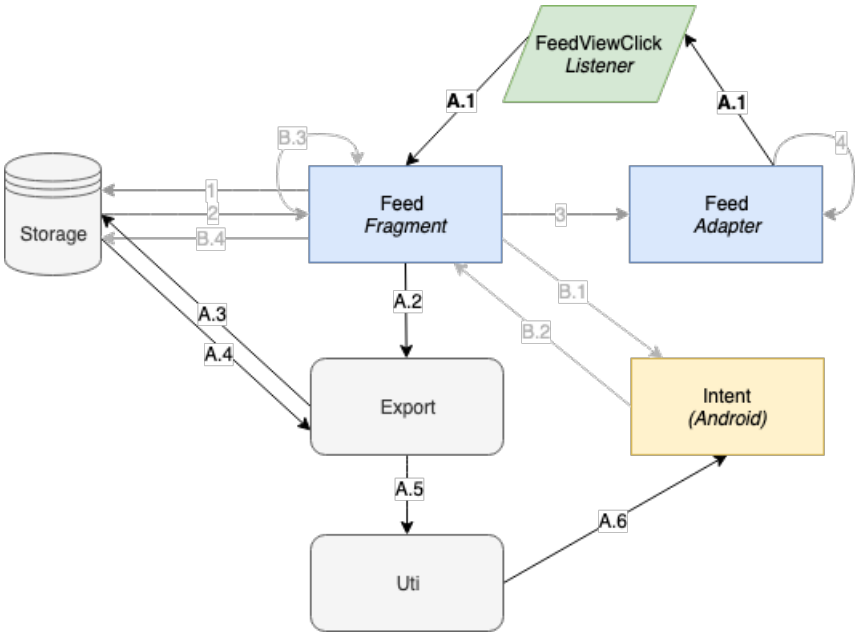

Figure 5.5: Implementation of sharing functionality (A): Exporting one or all Records

record information is sent to the `Feed Fragment` 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 `Feed Fragment` delegates record information to the `export` method inside of the `Export` class. The class is responsible for enabling exportation.

A.3 An operation to retrieve all samples related to the record with the use of the `SampleViewModel` 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 (Ref: Data Format). To enable the sharing interface provided by Android, 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 component uses the reference to the file location.

A.5 The encoded file is retrieved with the use of `FileProvider` (facilitates secure sharing of files [ref]). The code for this step are

```
1 static void shareFileIntent(Activity a, File file) {
2
3     Uri fileUri = FileProvider.getUriForFile(a.
          getApplicationContext(), a.getApplicationContext().
```

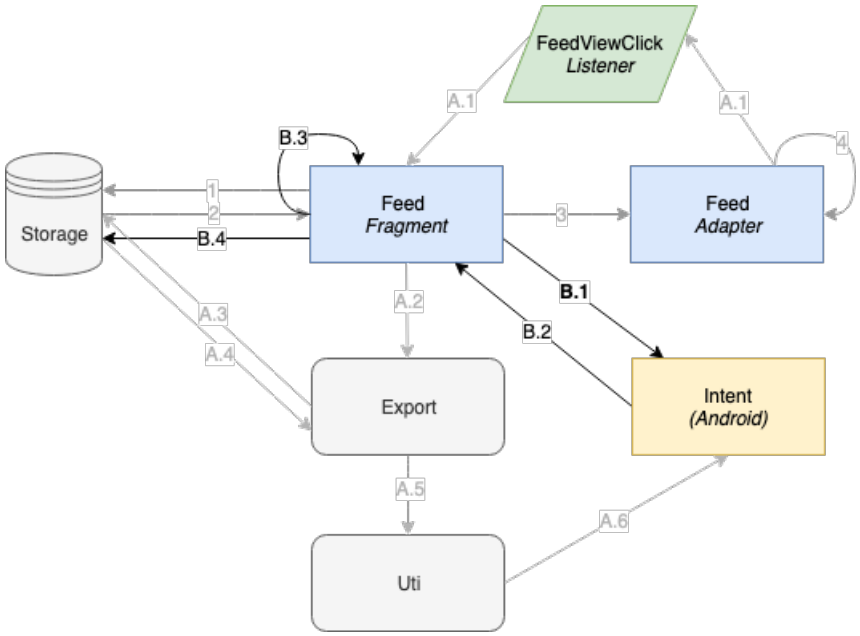

Figure 5.6: Implementation of sharing functionality (B)

```
        getPackageName() + ".provider", file);

    Intent iShareFile = new Intent(Intent.ACTION_SEND);
    iShareFile.setType("text/*");
    iShareFile.putExtra(
        Intent.EXTRA_SUBJECT, "Share Records");
    iShareFile.putExtra(Intent.EXTRA_STREAM, fileUri);
    ...

    a.startActivity(
        Intent.createChooser(iShareFile, "Share Via"));
}
```

Listing 5.3: My Caption

A.6 The user is displayed with a popup interface with several options to share the file over a media. An illustration of the layout can be found in Section Representation.

5.2.2.2 Action B: Import a Record from the Device

In Figure 5.6, an illustration of importing a record from the device is shown. Importing conists 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 users database. The steps can be narrowed down to:

B.1 The user requests to view the import record interface. The interface is provided by Android, and allows the user to select particular kind of data on the device (ref). The code for this action is:

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

Listing 5.4: My Caption

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

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 has to be done. After the absolute path is found, the data is decoded accordingly to the data format, and the records are sent back to `Feed Fragment`.

B.4 The necessary record information and the samples are extracted from the decoded data, and are inserted into the users database.

5.2.3 Modules

Modules are standalone application, that provides data enrichment and extended functionality to the application. The modules leverages the records and samples to analyze, evalutate or detect sleeping disorders. In order to add and launch a module in Nidra, we need the modules package name. The package name and the name of the module-application can be obtained in Android. Thus, the actions to enable modules in the application are: (A) add a module; and (B) launch a module.

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

In this Subsection, we will take a look into the steps that are taken to enable the 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 on beforehand. By listing through the installed application on the device, the user can select the desired module to be

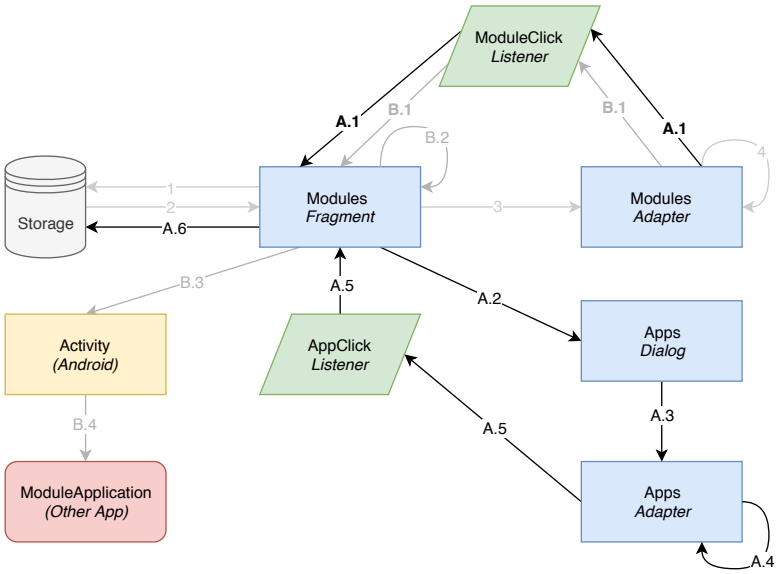

Figure 5.7: Implementation of module functionality(A): Add a Module

added in Nidra. In Figure 5.7, an illustration of adding a module is shown, and the steps can be narrowed down to:

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

A.2 The `Modules Fragment` lauches a custome Android dialog, which will list all of the installed application on the device.

A.3 The `Apps Adapter` will fetch all of the application that is not a system package, already installed module, or the current application (Nidra). Next, the 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 `Modules Fragment` through the callback reference `onAppItemClick` between these components are made. The callback contains an object with the `PackageInfo` 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 is stored in our database for modules through the DAO interface.

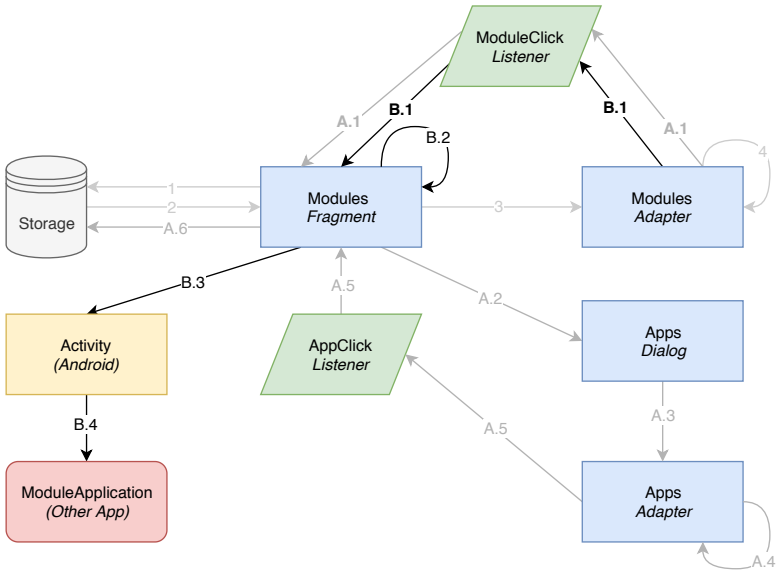

Figure 5.8: Implementation of module functionality(B): Launch a Module

5.2.3.2 Action B: Launch a Module

A module is launched in a seperate process, due to Android prohibits launching for other applications inside of an application. All added modules are listed and presented to the user in a separate screen, and on launch of a module, all of the data that Nidra obtains from recordings, are encoded into a JSON format and bundled with the launch of the module. In Figure 5.8, an illustration of launching a module, and the steps can be narrowed down to:

B.1 Upon an event for launching a module in `Module Adapter`, the packagename of the module is sent to the `Modules Fragment` through the callback reference (`onLaunchModuleClick`) between these components. The packagename will be used to launch the module-application.

B.2 All of the records and samples on the device for the user, is bundled and formated into a JSON, and launched:

```
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);
```

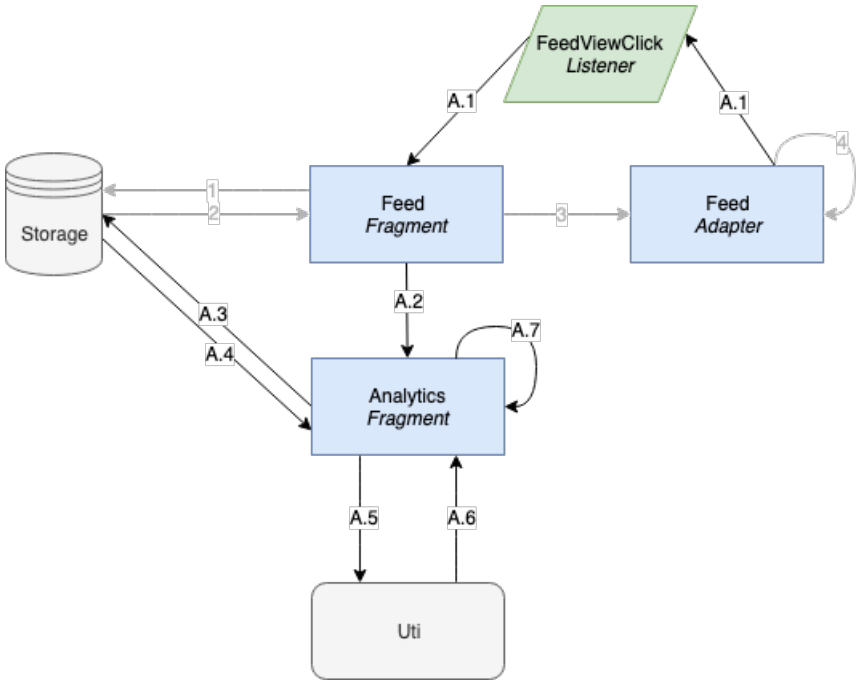

Figure 5.9: Implementation of analytics functionality (A): Display a Graph for a Single Record

```
10
11      moduleApplication.putExtras(bundle);
12
13      startActivity(moduleApplication);
14  }
```

Listing 5.5: My Caption

B.3 The activity uses the data provided in the `Intent` that includes the packagename (the name of the module-application to determine the correct application).

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

5.2.4 Analytics

Analytics is the part of illustrating and analyzing the records. In Nidra, the analytics part of the implementation is limited to a time-series graph for a single record. However, there are possibilities of extending the `Analytics Fragment` with other graphs based on the current structure. The current action for analytics is A) display a graph for a single record to the user.

Similar to sharing, the records from the database have to be presented. The `Feed Fragment` contains a `RecyclerView` which populates the records

into inside the `Feed Adapter` (steps: 1-4). The adapter contains all the interactions and the event handling (i.e., button event listener for analytics) for a single record.

In this Subsection, we will take a look into the steps that are taken to enable the action.

5.2.4.1 Action A: Display a Graph for a Single Record

Nidra provides a simple time-series graph of respiration data obtained during the recording. The graph data is plotted into a Library, which enables interactions (e.g., zoom and scrolling) on the data. The X-axis is the respiration value based on the Y-axis time of sampling. In Figure 5.9, an illustration of displaying a graph shown. The steps can be narrowed down to:

A.1 Upon an event for analytics on a selected record in `Feed Adapter`, the record information is sent to the `Feed Fragment` 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 `Analytics Fragment` is created, and an transition from the `Feed Fragment` to the `Analytics Fragment` is made. Alongside, the record information is transmitted.

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

A.4 The `Analytics Fragment` retrieves all of the samples related to the record. The samples has to be structured according to the graph library to display an interactive time-series graph. (GraphView (ref)).

A.5 Each sample has to be extracted from the sample-data, acccordingly to the sensor data structure.

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

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

5.2.5 Storage

Storage facilities persistent data which remain available after application termination. In Nidra, there are four individual data entities (i.e., record, sample, module, and user). In the Subsection about Data Entities, we discussed the design choices of the individual data entity. The standard

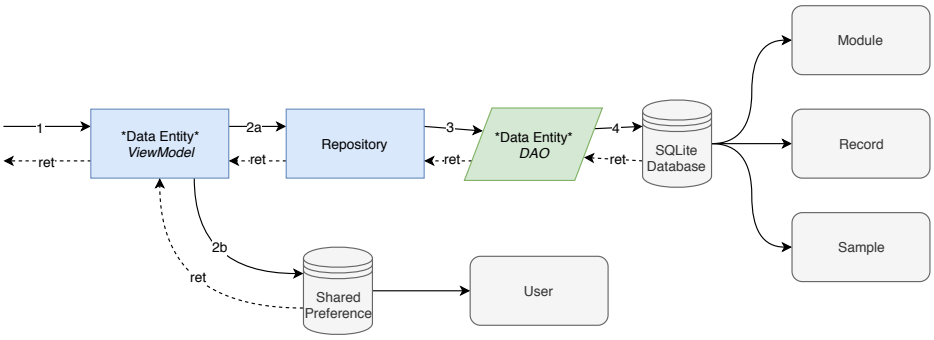

Figure 5.10: Entity Relationship Diagram

CRUD operations on each data entities are: insert, update, and delete. Also, the user has an operation to retrieve the biometrical data, module and record have an operation to retrieve all of the entries in the database, and samples have an operation to retrieve all of the entries corresponding to a record.

Android Room provides an abstract layer over SQLite to enable easy database access [Cite]. In Figure 5.10, the flow for accessing and retrieving the data from the database based on the Android Room architecture is shown and can be described as:

1 Each data entity has a `ViewModel` where all of the CRUD operation (e.g., insert, update, delete, or retrieve) 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) [CITE].

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) [CITE]. 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 methods to read and write. 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 accessed and retrieved from the SQLite database.

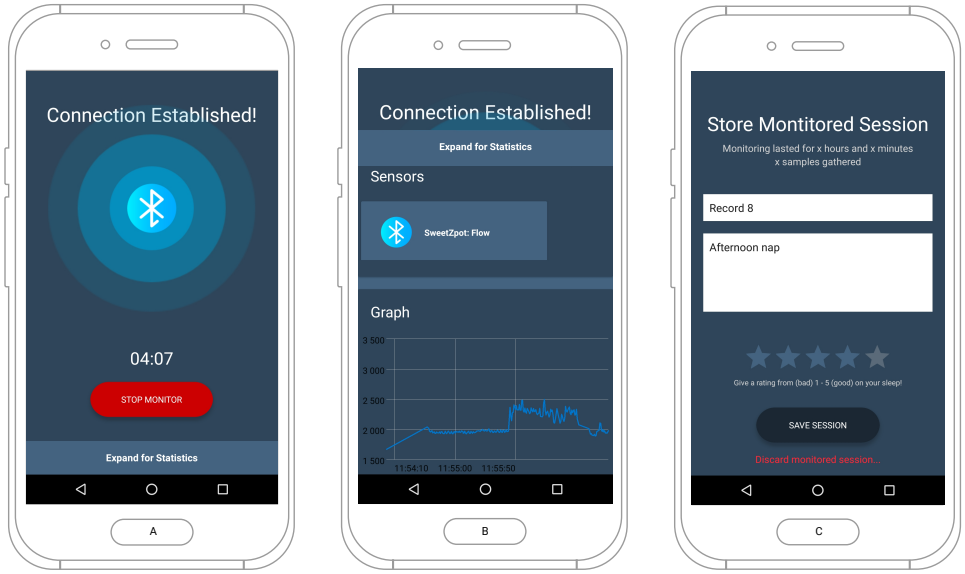

Figure 5.11: The recording screen displayed to the user; with the screen: (A) during a recording, (B) real-time analytics, and (C) finalizing the recording

5.2.6 Presentation

In this Subsection we will present the user-interface (UI) based on the functionality of recording, sharing, module and analytics. The interface is developed based on the design descisions of the application, 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.

5.2.6.1 UI: Recording

Figure 5.11 shows the screen for: (A) during a recording, (B) the real-time analytics, and (C) finalizing the recording.

A The screen has a ripple effect to indicate the state of the recording to the user. There are two types of ripple colours; a blue ripple effect for samples acquistion, and a grey ripple effect if the sensor has disconnected. The ripple effect is only active if the screen is turned on, in order to perserve battery life. During disconnects between the sensor and the device, the user provide no extra input to resolve the issue. However, if the numbers of attempts is reached, the user is presented with the finalizing screen (C). Also, this screen display the elapsed recording time, and has a visible button to stop the recording.

B The user can extend the interface for viewing statistics regarding the recording session. Currently, the interface lists all of the available sensor sources, also a real-time interactable graph of respiration.

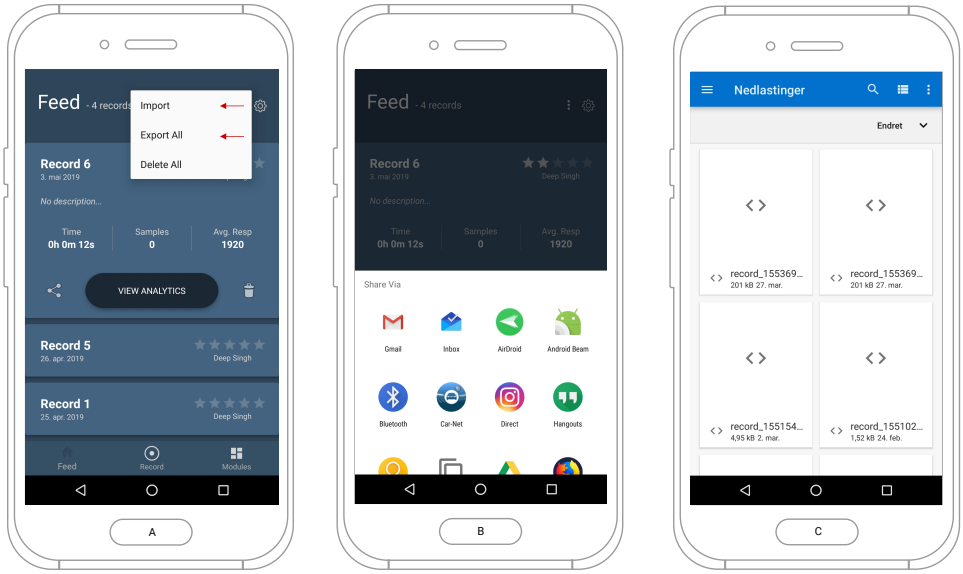

Figure 5.12: The sharing screen displayed to the user

C The finalzing screen allows users to specify the title and description of recording, to make it distingushable, in addition to giving a feedback on recording so the doctors/researchers can review it. Also, the user can give a rating between 1–5 (where 1 is bad, and 5 is good) to rate the sleeping.

5.2.6.2 UI: Sharing

Figure 5.12 shows 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. The user can choose to export one single record, or export all.

B By pressing export all, an overlay with a Android provided sharing screen is presented, where the user can choose a media to export the file on.

C By pressing import, an overlay with downloaded files on the users device is presented. The user can press on the desired file, and the file will be parsed and added to the users collection of records.

5.2.6.3 UI: Modules

Figure 5.13 shows the screen for: (A) list without any modules, (B) list of installed application, and (C) list with modules.

A The modules screen is shows all of the added modules to the users.

Figure 5.13: The module screen displayed to the user

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.

5.2.6.4 UI: Analytics

Figure 5.14 shows the screen for: (A) single record in the list, and (B) the analytics for the record

A The user can expend records in order to view more functionality and statistics of the record. One of the functionalities are to view the analytics of the record.

B An interactable graph is presented to the users, and the graph is populated with the samples obtained from the recording for selected records. The graph shows the respiration value (Y-axis) on given time (X-axis) of sampling.

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 perserve battery time. However, this mechanisms is not viable for collecting data over an extended time, because it can kill our applications

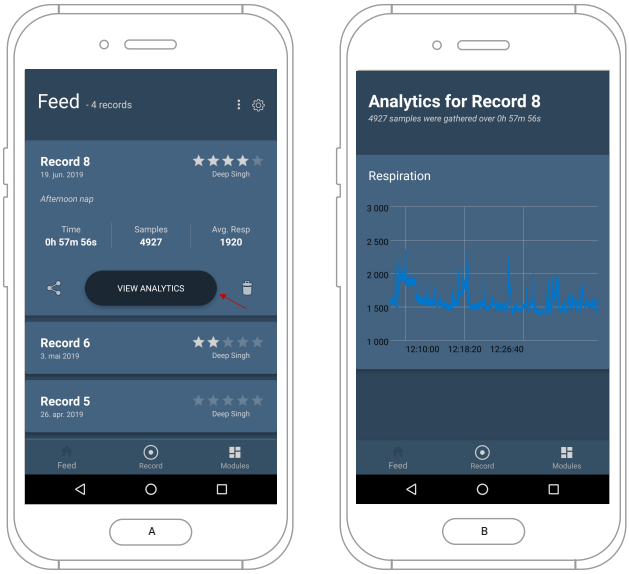

Figure 5.14: The analytics screen displayed to the user

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 documentations states that holding onto a wake lock for a longer period of time, shortens the devics battery time. Therefore, it is important to release the lock when the recording has terminated. In order to use wake locks, the permission has to be added in the application's manifest file. Nidra utilizes the wake lock when the recording has started (inside of the `onCreateView`), and can be seen as in: (LISTING X). Also, the lock is released when the activity is destroyed by terminating the recording session.

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

Listing 5.6: My Caption

5.3.1.2 Priority

Process's lifecycle is not directly related with the application itself; however, determiend by the system detecting parts of applications that are running, how important they are to the user, and how much memeory 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 certain measures to prolong the servives run time. That is, to increase the process importance in the "hierarchy". By forcing the process to be a *foreground process*, we can for the most cases prevent the system to kill the process. We can create a foreground process by creating a starting a foreground process.

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 consituted by application components (e.g., activity, provider, broadcast receiver, and service), and each component contains meta data describing the application component. Also, all of the components has must be declared in the manifest with the reference to the component in the name (`android:name`). The application components were presented in the Section X, and below we describe the permissions and a few application components that are of interest to us.

5.3.2.1 Nidra

The Nidra manifest file consitute of four activites, one service, and one provider. The latter is used to share a record between applications. Providers enbles access to between other applications that wish to access a file or data of our applications. With the provider, an direct URI link that passes throught the provider, grants a more secure sharing of data between application. In the Listing X, the attribute `authorities` is the name that identify the data offered by the provider (often distingushed 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>
```

Listing 5.7: My Caption

As for Nidra, the permission are presented in Listing X. The noteworthy is the wake lock permissions and the permissions to store data in external storage, and the interal storage.

```
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" />
```

Listing 5.8: My Caption

5.3.2.2 Flow Sensor Wrapper

As for the Flow sensor wrapper, the application components structure are based on the templated created by Gjøby. With an expection of the activity and the service for Bluetooth. However, the permissions are the interesting part of the manifest. In order to leverge 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 available sensor source in the area. Without the permission, Android do 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 [...]
```

Listing 5.9: My Caption

Part III

Evaluation and Conclusion

Chapter 6

Evaluation

In this chapther, experiements are performed in order to evaluate the application based on the requirements defined the problem statement:

1. The application should provide an interface for the patient to 1) record physiological signals (e.g., during sleep); 2) present the results; and 3) share the results.
2. The application should provide an interface for the developers to create modules to enrich the data from records or extend the functionality of the application.
3. The application should ensure a seamless and continuous data stream, uninterrupted from sensor disconnections and human disruptions.

The experiements are designed to test the application in a crowded environment, a extended recording to measure battery usage, user test the application, and creating a simple module. When evaluating the experiments, the observations and the results are the most interesting metric to us, because it gives us a heads up on whether the design (Chapther 4) and implementation (Chapther 5) of our perspection meets the demand and persieved (the application is adequently accomplished). Moreover, the feedback on the experiments allows for discussion on improvements that can be made, which is part of the future work on the on the application.

6.1 Experiments and Measurements

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

The experiment was conducted in collaboration with master student Joachim Dalgard at the University of Oslo at the faculty of *RITMO: Centre*

for Interdisciplinary Studies in Rhythm, Time and Motion.

The goal of the experiment was to analyze musical absorption, which is a state an individuals ability and willingness allows 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, we 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 around 50 minutes at Oslo Concert Hall on third of April and fourth of April 2019.

However, the motivation for this experiment for our application can be summarized into: (1) to test the application in a real-life and crowded environment, in order to analyze whether other mobile devices interfere or obstruct with the signals between the collecting sensor and the device; (2) to test whether the samples gathered are meaningfull, 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 kit to measure respiration during the concert. RITMO had multiple Flow sensors for disposal; however, they had no suitable mobile application that could record with these sensors. Also, with their equipment they experienced that Flow sensor kits tended to disconnect every 10-15 minutes, resulting in fragmented recordings for a single session. Therefore, they reached out to *Insitute for Informatics* in hopes of a solution. Our application was a suiting match for both parties, hence a collaboration was formed [? do i need this]. We arranged for six Android devices and reached out to the participants to bring their Android devices if they had one. 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 most of the attendees had a mobile device, and a few shares of them had BlueTooth activated on the device. Based on these estimates, 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 application. 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.

Model	Samsung Galaxy S9	OnePlus 3T	Google Pixel XL
Operating System	Android 8.0	-	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

6.1.1.1 Preperations

In order to partake in the experiments, we had to ensure that the participant had the sensor placed accurately on their body and the mobile devices were configured correctly. Additional tests were also performed in order to ensure the application worked as intended, and also to prevent any unforeseen events or bugs that could have occurred during the recording.

Device Configuration The device models in our disposal had to be configured with the applications to enable recording on Nidra. First, the data stream dispatching module was installed on the devices. Second, the sensor wrapper for the Flow sensor kit was configured with one Flow sensor on beforehand—in order to reduce the time to set up the mobile device and the sensor on the participant before the concert. Lastly, the Nidra application was initiated with a unique name (A-F), as such to distinguish each participant with the sensor and device model. In Figure 6.1, the device models are listed with their specifications and the OS ... [Skriv mer]

Body Placement The respiration 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.

6.1.1.2 Results

We gathered all of the recordings from the various devices by using the sharing functionality in the application and sent it to our application. 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 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 (breathing) 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. The graphs are plotted with Python and the library

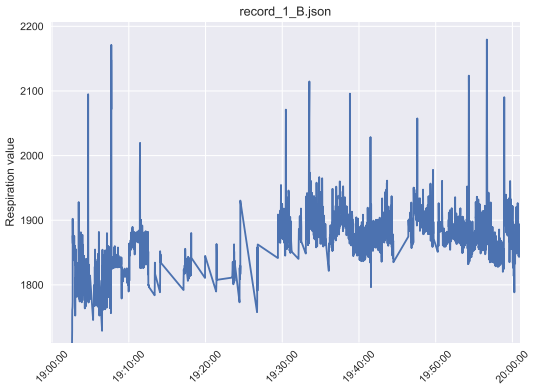

Figure 6.1: Concert Day 1—Mobile Device B

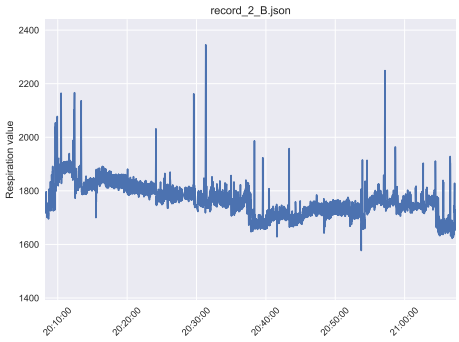

Figure 6.2: Concert Day 2—Mobile Device B

MatPlotLib in order to analyze and evaluate the sample values; however, an identical graph is also presented in the application.

From the thesis "..." by Løberg, 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 in 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 the 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 noticably deep breathing throughout the recording, with a few shallow breathings at the beginning. Conclusively, both Figures shows no nuances for no breathing during the recording. However, there are signs of normal breathing, with few instances of shallow and deep breathing.

6.1.1.3 Analysis & Discussion

We will discuss all twelve records by analyzing the respiration values. 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 & Roof Samples: 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. & Roof Samples: 4288

The recordings for the six devices for the two dates are represented in Table 6.2 and Table 6.3. Each table exhibits data that is extracted from each recording from the mobile device and can be characterized as:

Roof Sample 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).

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

Loss Count The number of missing samples based on `Roof Samples`. This can be calculated as:

$$Loss\ Count = Roof\ Samples - Samples\ Count \tag{6.1}$$

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

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

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

Disconnection Time Is the accumulated time of disconnection.

The device models A, E, and F are noticeably accurate (with some noises which presumably occurred during parsing), there are no apparent disconnects occurred during the recording for these models both of the days. In contrast, device model B, C, D shows an outburst of disconnections during recording. Especially, model D has high loss percentage both of the days, which is reflected in the disconnection time. Also, model B had a high loss percentage on the first day; however, significantly less loss percentage the second day. Similar resemblance can also be seen in model C.

We could conclude on the fact that some mobile device models or some sensors is malfunctioned, and for this reason we see higher loss rate. Unfortunately, with insufficent data to point out whether that is correct, is is something we cannot conclude on. Moreover, [Skriv mer om Android OS version].

6.1.1.4 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 senor 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 samples from the recording, it is identified that the Flow sensor has a tendency of disconnecting, but the application managed to provide a continuous data stream by reconnecting with the sensor during recording. In the end, the records were successfully shared across applications, which enabled us to analyze the recordings.

To conclude, the goal of the experiment was to analyze musical absorption during a concert, but that is not the motivation for our application. However, we provided an application that collected the data during the concert. Thus, the collected data can then be used to conclude a correlation between breathing and musical absorption. Also, inconsistency in the sampling of the recordings makes it difficult to determine the source of the problem. We could argue that the analysis of the aggregated data in the tables might indicate that the sensor or the mobile is a malfunctioned, however, due to insufficient data this conclusion cannot be drawn. Although, the application managed to collect data with the use of the Flow sensor kit, in an environment filled with other mobile devices that could have interfered with the signals, as well as reconnecting to the sensors during the recording. [Reflect more on the motivation goals].

6.1.2 Experiment B: 8-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 drainage during this period. The experiement is conducted to demonstrate that the appication

is suited for collecting data over an extended period. In most cases, an 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 perspection of the applications outcome to the user. Therefore, in this experiment we are checking for the battery consumption on the mobile device, as well as the Flow sensor kit.

6.1.2.1 Description

The experiement was conducted by strapping the Flow sensor around the throax (just below the armpits), configuring the the Flow sensor wrapper to connect with the Flow sensor, and pressing the "record" button in Nidra. The device and sensor started with 100%. Also, the device was disconnected from WiFi, and left stantonary and unattended during the period of recording.

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

6.1.2.2 Results

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

$$Battery\ Consumption = \frac{(Start\% - End\%) * Device\ Capacity\ (mAh)}{Recording\ Duration\ (Hours)} \quad (6.3)$$

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

6.1.2.3 Conclusion

6.1.3 Experiment C: Performing User-Tests on Two Participants

One goal of the application is that it should be user-friendly enough for patients to use and understand how the functionality of our application

works. To evaluate whether this goal has been achieved, we found two participants that agreed to partake in the experiment. One of these participants is not proficent in the use of modern technology, while the other participant has a lot of experience. For simplicity, we will refer to the participant as participant `A` and `B`, respectively.

The testing process consisted of three parts: a presentation of the application, tests to record and share the recording, and an interview. The tests were not performed during the night, due to ... However, this tests focuses on user-friendliness, rather than recording over an extended period. The participant was informed with the scinario the application was designed for.

Presentation We presented the goal and the functionality of the application to the participants in a simple and intuity way. However, we mainly focused on explaining the recording functionality and the methods of sharing the recording across applications. The presentation was took around 10 minutes, including questions and clarifications.

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

Interview The interview was conducted after thes tests were performed, so the participant had their feedback fresh in memory. We structured the questions in the interview accordingly to the PACMAD methodology.

6.1.3.1 Testing

To evaluate whether the application presents the functionality adequently 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 check for connected sensor sources, and the graph for the sampling

T3 Stop the recording, and finialize the recording session by giving it a name, description and rating.

T4 Find the recording in the feed of recordings.

T5 Share the recording over e-email.

T6 View the analytics for the recording by interacting with it (e.g., zooming and scrolling).

For this test to be successful, the participants must for all of the tasks be able to correctly identify the actions. There were no interference or guidance on how to perform the tests, besides guiding the participants into performing each task. These tests were designed to observe and simulate the primary actions the user mainly focuses on in the application. Thus, the functionality of modules and importing recordings were left behind.

6.1.3.2 Observations

Participants A—less technical—had to familiarize with the setting of the application, thus had a hard start to begin with. It took the participant longer than exepected to perform T1, however, managed to proceed on recording and was bit uncertain whether the recording had started or not. Also, 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 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 find the recording on the feed screen. However, the T5 and T6 was bit unclear to the participant, because the functionality is hidden, and in order to find the actions you have to press the record to expand the information. However, after that was clear, the participant managed to perform both task sufficently.

Participants B outperformed participant A, due to the participent were more fimilar with technical systems. The participant managed to start the recording quickly, however, were unsure whether the recording had starting (clicking around on the interface for feedback), untill the ripple effects appeared on the screen. Similar to participant A, participant B clicked on the interface for statistics, rather than swiping. Also, had a hard time to interact with the graph due to the interface moving. The particpent B managed to perform T3-T6 with ease and without any interupts or objections.

6.1.3.3 Discussion

Conclusively, the tests were not performed to its desired intent, due to the fiddeling in on the recording screen. However, the participant managed to perform most of the tasks regardless, albeit the task T1 and T2 were hard to comprehend.

Both of the participants had a hard time to understand whether the sensor was collecting data, despite the ripple effects on the screen. Argueably, the user can familiaries with the state of the recording and the ripple effects which indicates sample acquestion, however, to a new user

that is bit hard to time understand this. The source of this problem is that the sensor takes some time (i.e., 30-60 seconds) to output the samples, however, the sensor establishment between Nidra and the sensor wrapper occurres imediately.

Also, the interface to view for expanding the statistics was bit tedeous to understand.

Besides this, there we no noticably complains by the participant. They found the color scheme of the application to be smooth and fitting, and the application to be moderen. Also, they find stuff to be well organized and were not drowned in selections and actions to perform. All of the functionality is restricted to few and simple actions per screen.

6.1.3.4 Survey

The interview questions were performed after the tasks. The questions followed a PACMAD survey structure, which is a model to identify the usability attributes and are structed into: effectiveness, efficency, statisfaction, learnability, memorability, errors, and cogitive load. We created a survey based on some of the structure, however, we were unable to perform any cogitive load or memoriability tasks during the testing. Hence, the participant filled out the questions regarding:

Effectiveness & Efficency

- What were your inital thoughts on the application

  Participent A: The application looked nice and simple-

  Participent B:

- How difficult was it to start a recording (very hard (1)–very easy (5))

  Participent A: 1

  Participent B: 3

- How would you rate the feedback you got during a recording (very bad (1)–very good (5))

  Participent A: 2

  Participent B: 3

- How difficult was it to stop a recording? (very hard (1)–very easy (5))

  Participent A: 5

  Participent B: 5

- How difficult was to browse/find previous recordings? (very hard (1)–very easy (5))

  Participent A: 3

Participent B: 5

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

  Participent A:

  Participent B:

Satisfaction

- How satisfied were you with the "journey"? (not satisfied (1)–very satistifed (5))

  Participent A: 3

  Participent B: 3

- How satisfied were you with this application overall? (Not satisifed (1)–very satisfied (5))

  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? (Not so friendly (1)–Very friendly (5))

  Participent A: 4

  Participent B: 4

- How would you rate the color palette of the application? (Not so pleasing colors (1)–Very pelasing colors (5))

  Participent A: 5

  Participent B: 5

- How would you rate the general layout of the application (buttons, text, navigation, etc...)? (Very bad (1)–Very good (5))

  Participent A: 4

  Participent B: 5

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

  Participent A:

Participent B:

6.1.3.5 Conlusions

6.1.4 Experiment D: Creating a Simple Module

One of the goals of the application is to provide an interface for the developers to create modules, which allows for data enrichment and extended functionality. 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 was for the participant to create a new module in our application, which utilized the records in order to find the record that has the highest number of samples. The development of a user-interface was not evaluted, thus displaying the correct answer on the screen was sufficent.

Before the the participant started on creating a new module, an simple introduction on the concepts and data was performed. Mainly, the fact 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 up on launching the module-application (with Intent). The data structure were also introduced, i.e., an array of records with meta-data and correlated samples.

6.1.4.1 Process

The participent created a new Android project named *TestModule*. For the simplicity, all of the development occurred in one activity. The first task was to get the JSON string that was bundled on launch in an Intent with the key of *data*. The second task was to decode the data, the participant used `Gson` library in order to parse the data accordingly to the struture. And the final task was to display the results, which the participant did by iterating through the records and finding the one with the highest sample count.

6.1.4.2 Observations

Despite the tasks seemlingly looks easy of making a simple module, the participant had a rough start. The participant was aware that module-application received data in a bundle, and the extraction could . 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 (while supplies the module with the data). Albeit, the participant managed to handle this error early.

The next challenge for the participant was to understand how to transform the JSON string into valid objects in Java. First, the participant had to create an object which encapsulates the record and and a list of samples (`ExportObject`). Then, three seperate objects (i.e., record, sample, and user) had to be created, that were identical to the payload. As such, the participant could decode the JSON string into an list of the `ExportObject`, and retrieve the necessary data.

Once the participant had all of the data, the rest of the tasks was simily accomplished. THe participant iterated through the list of records, and found the one with the highest sample count, and displayed it on the screen.

6.1.4.3 Results

In Figure X, the module-application is illustrated. As of the experiement, there were five records on the mobile device—and the highest was the record named `Record 17` with `163` samples—thus, the experiement was successful conducted. It took the participant approximately 30 minutes to develop this module-application, where most of the time went to parse the data.

6.1.4.4 Discussion & Conlusion

Conclusively, the participant managed to create a module, with some hurdles in the way. Noticably, while compling the project the participant had to open its module through the Nidra application in order to get the data. There are two ways to prevent this from happening, one way is to establish an IPC (discueed in the design chapther) with the use of an AIDL file and output the data from Nidra to the application through the IPC connection. The second way is to cache the data in the module-application, for temporary storage of the data. However, both methods increases the complexity of the module-application. Nonetheless, the former improvement can be made in Nidra.

The second noticable occurances, was the the parsing of the JSON data. JSON is a bit tedeous to work with in Java. As of now, there is no direct support for parsing JSON in Java, hence third-party libraries has to be used in order to do so. To combat the start-up face of a new module-application, we will include boilplate code (found in Appendix C), to make module implementation easily comprehenceable for future developers.

6.2 Main Findings

Any changes commited in the module or Nidra, is not reflected with each other. When changing the code in the module application, Any changes

commited in the module or Nidra, is not reflected with each other. When changing the code in the module application, Any changes commited in the module or Nidra, is not reflected with each other. When changing the code in the module application,

Chapter 7

Conclusion

7.1 Summary

In this thesis, we designed an application for recording, sharing, and analyzing breathing data with the Flow sensor kit. The motivation of this thesis was to extend the CESAR project by providing an interface for patients to collect data during sleep. The data would aid researchers/doctors to analyze, diagnose, or examine the patient for sleep-related breathing disorders (e.g., Obstructive Sleep Apnea).

In the design of this thesis, we identified the tasks to meet the system requirements of this application. Based on the design choices, we implemented an Android application called Nidra. The application leveraged the tools provided by the previous work performed on the CESAR project, in order to manage current sensor sources, as well as future sensor sources. We implemented a sensor wrapper for the Flow sensor kit, which connected with BlueTooth LE for data acquisition. During the recording, the samples from the sensor sources were unpacked and stored in our SQLite database. Also, we ensured for continuous data stream—uninterrupted from sensor disconnections and human disruptions—by reconnecting with the sensor source. At the end of the recording, metadata about the recording—including the state of the user's biometrical data—was stored in the database. Previous records were presented to the users, where each recording had the functionality to view the analytics or share the record. The sharing functionality allowed for transmitting records across applications, such that researchers/doctors can view and analyze the patient's record. The analytic part of Nidra constituted of a time-series, which enables the users of the application to interact with the data from the recording. Conclusively, while Nidra's design is for recording sleep with various sensor sources, the application can be applied to other fields (e.g., recording respiration during a workout).

In the field of detecting and diagnosing sleep-related breathing disorders, we could see that related work collects physiological data on

various method to predict or examine sleeping disorders (e.g., Obstructive Sleep Apnea). Therefore, by creating an extensible application which can centralize the application of various techniques and methods in this field, they would benefit by creating an extention to our application in order to save time and effort in creating a base application of their project. As such, Nidra provides an interface for future developers and researchers to create modules which enrich the functionality of Nidra or provide data enrichment to the patient's recordings. The modules are independent Android applications, which is installed alongside Nidra and utilizes the provided records by Nidra. For example, a module could be to use the data on the patient's device to feed a machine learning algorithm that predicts or diagnose the sleeping disorder.

Finally, we performed experiments on the application to determine whether the system requirements were fulfilling. The experiments ranged from collecting respiration data for analysis of musical absorption, testing the application over an extended period, creating a simple module-application, including testing the application on real users. The results and observation made during the testing, allowed us to gain a broader understanding of the application, as well as improvements that can be made–discussed in future work.

7.2 Contribution

In the problem statement of this thesis, we defined three research goals. In this Section, we restate the goals in together with how our work solves the goals.

Goal 1 *Integrate and support for Flow sensor kit with the extensible data acqusition tool*

In the implementation chapther of this thesis, we created a sensor wrapper which were integrated into the Data Stream Dispatching Module. The Flow sensor kit uses BlueTooth LE as an communication channel, thus, we used the APIs provided by Android to connect with the sensor. We extracted the data from the sensor, noteworthy the respiration data, as well as the battery level, mac address and firmware number.

Goal 2 *Research and develop a user-friendly application which facilitates collection physilogical data through the extensible data acquisition tool.*

Through the course of this thesis, we have discussed the design of an application which facilitates the collection of respiration data, with the focus on the environment of use. The application is of most likely to be used during the night and early morning. Thus, selecting a color pallette that is not eye straining were picked. Also, we focused on limiting the number of actions that a user can perform on each screen. As previously mentioned, the application allows for collecting of

data over an extended period of time with sensor sources that are integrated with the tools provided.

Goal 3 *Create a "platform" solution for developers to create modules.*

As the field of collecting data

7.3 Future Work

With the exploration performed in design and experiments, we can see that the application can be enhanced or improved. While the application fulfilles the goals defined in the problem statement, there are still some improvement that can be made. Below, we present the a short description of future work alongside with our proposition of improvement.

Improve the user-friendliess of Nidra: In retrospect to the experiment, we could see that the participents 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 more informative description of the various states (e.g., connecting, recording, or disconnected).

Add support for other physilogical data The main focus in this thesis, was to gather breathing data during sleep. However, the possibility to extending Nidra to support other physilogical data is possible (e.g., hearth 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 soley 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 respository for shared records. Thus, by providing an interface for the patient to select a desired recipient, would allow for a simpler and convenient sharing for both the patient and the research/doctor. However, the implications are that the user's data are stored in a server, and without any security or authentication is prone to data leaks. That would be a risk in regards to the GDPR compliance.

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 the simplicity, this is sufficent, however, for future modules that depends on analyzing the data in real-time, that is not an applicable solution. Therefore, by establishing an 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 respectivly.

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 eliable to the user.

Appendix

Appendix A

Source Code

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

A.1 File and Folder Structure

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

- **Dispatcher**: contains the code for communication with the DataStreamDispatchingModule including the service for data acquisition, and the code for reconnectivity 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 connection with SQLite (with Android Room).
- **Utils**: comprise of miscellaneous code ranging from functionality to extract Flow payload to logic behind the export functionality.
- **View**: include the activities (discussed in Chapther 5) with seperation of various views (e.g., feed, module and recording).
- **ViewModel**: exposes the operations that can be performed on the data entities.

DataStreamDispatchingModule encompasses the implemention made by Bugasjki. However, the application was extended during the course of this thesis and can be reflected into 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 packagename, and stored in a `SharedPreferences` if it does not exists.
- **Sensor** is the sensor objective that is stored into the `SharedPreferences` mentioned above, with the name and the packagename of the sensor wrapper.

Flow encompasses the driver for creating a sensor wrapper made by Gjøby. The sensor wrapper adds the support for Flow sensor kit, and the extention 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 kit 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

Expatiate on Concert Experiement

B.1 Concert Day 1: Time-Series Graph

This Section illustrates the samples collected during the concert the concert on third April of 2019, which lasted for one hour. There are six indiviual records for this day:

B.2 Concert Day 2: Time-Series Graph

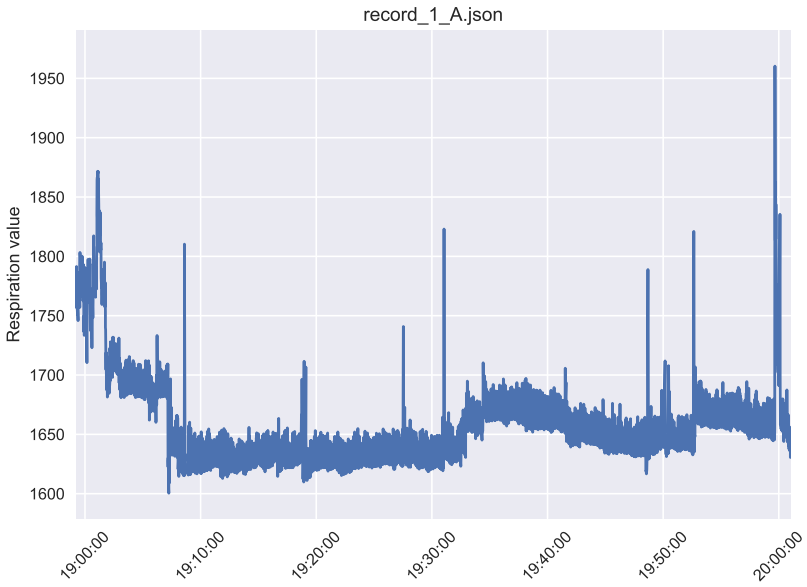

Figure B.1: Concert Day 1: Device A

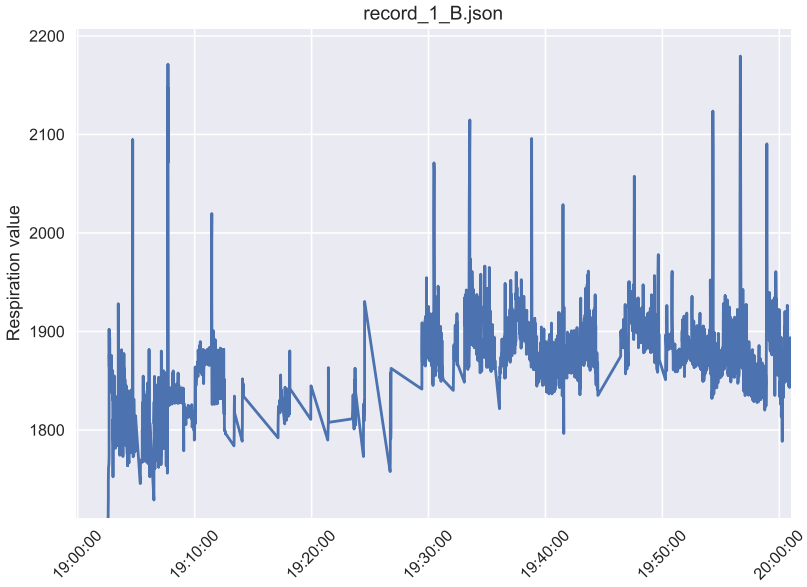

Figure B.2: Concert Day 1: Device B

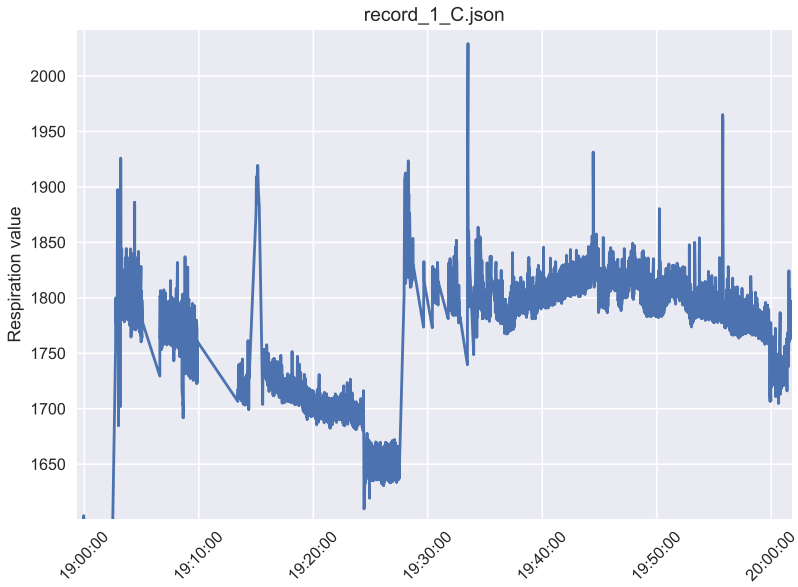

Figure B.3: Concert Day 1: Device C

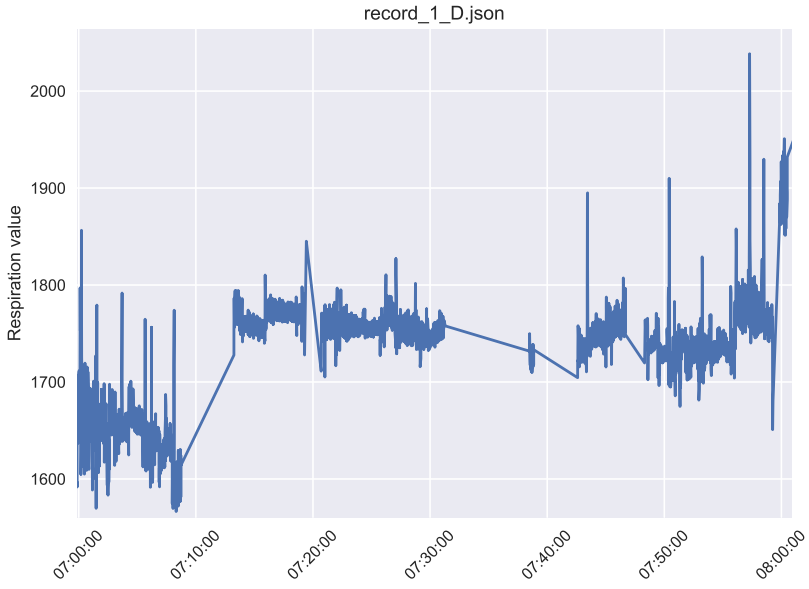

Figure B.4: Concert Day 1: Device D

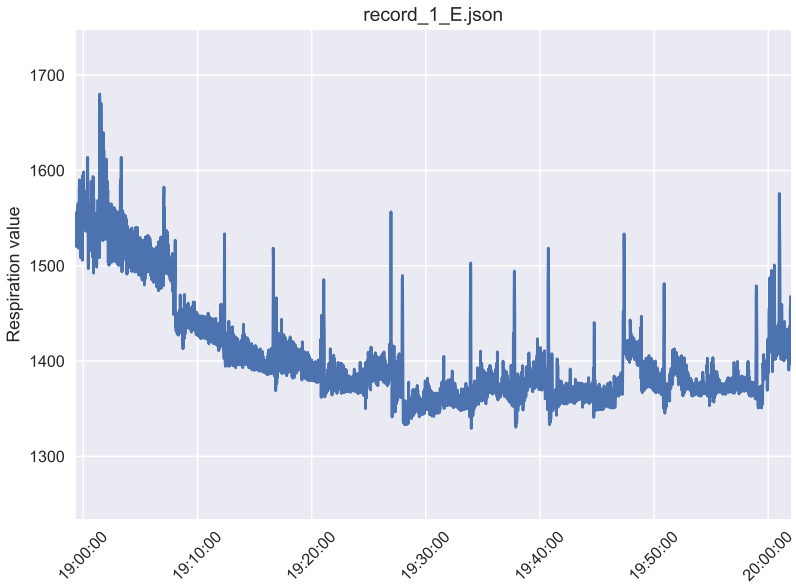

Figure B.5: Concert Day 1: Device E

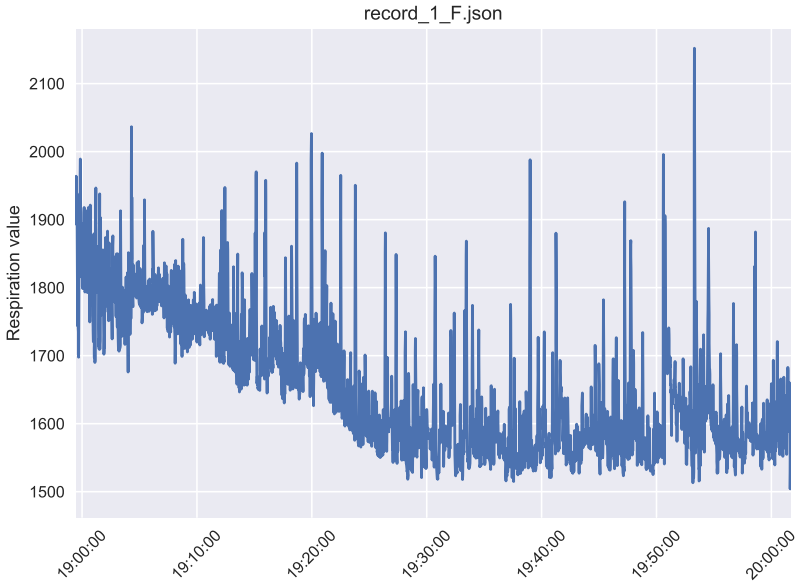

Figure B.6: Concert Day 1: Device F

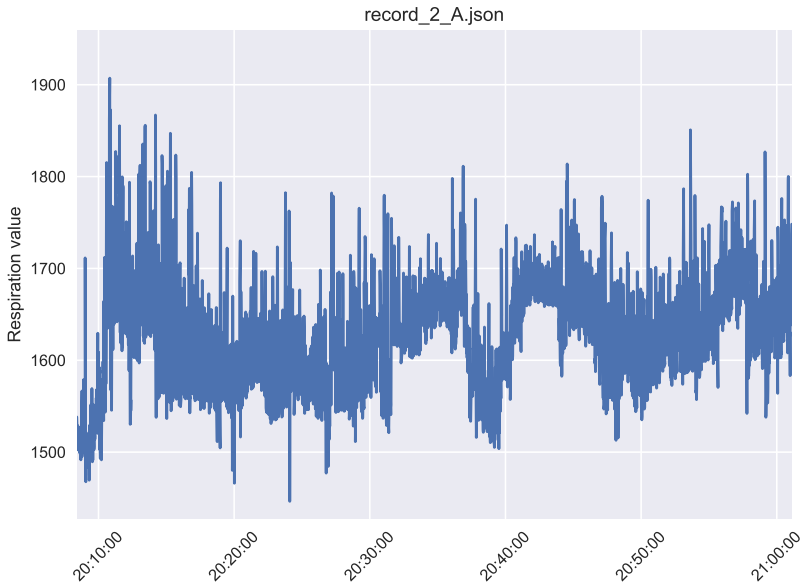

Figure B.7: Concert Day 2: Device A

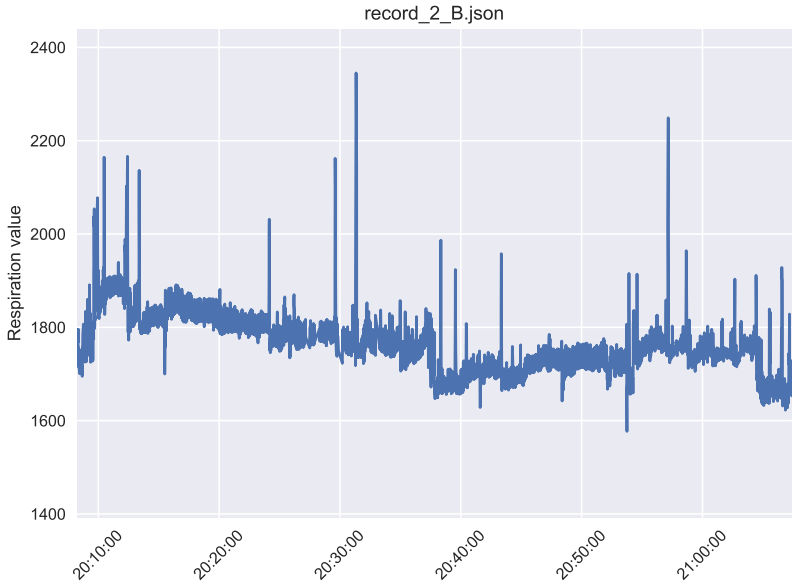

Figure B.8: Concert Day 2: Device B

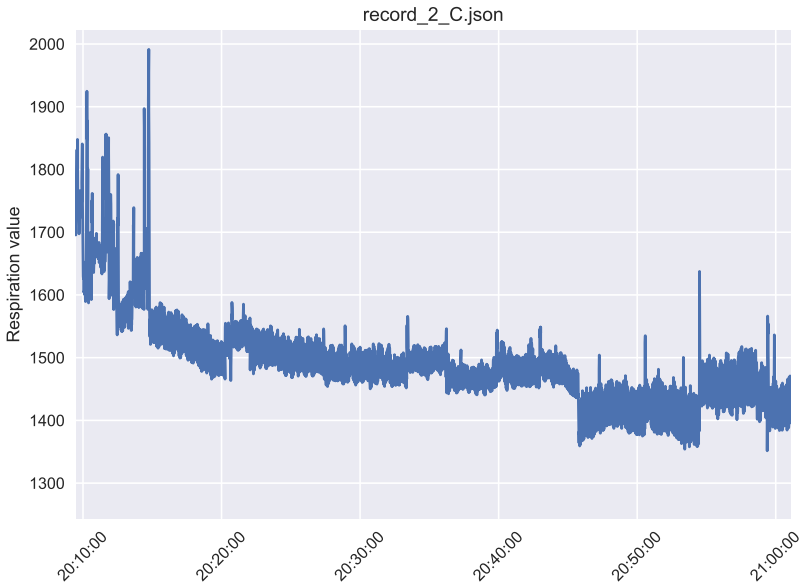

Figure B.9: Concert Day 2: Device C

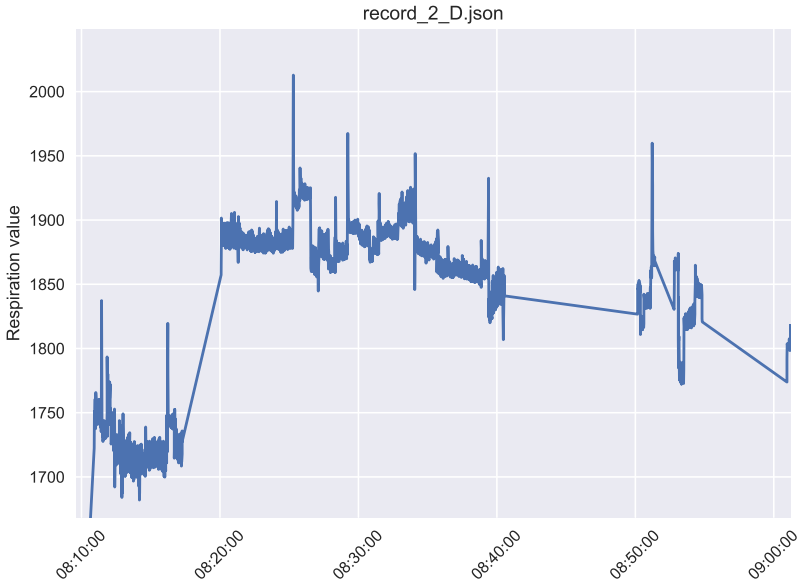

Figure B.10: Concert Day 2: Device D

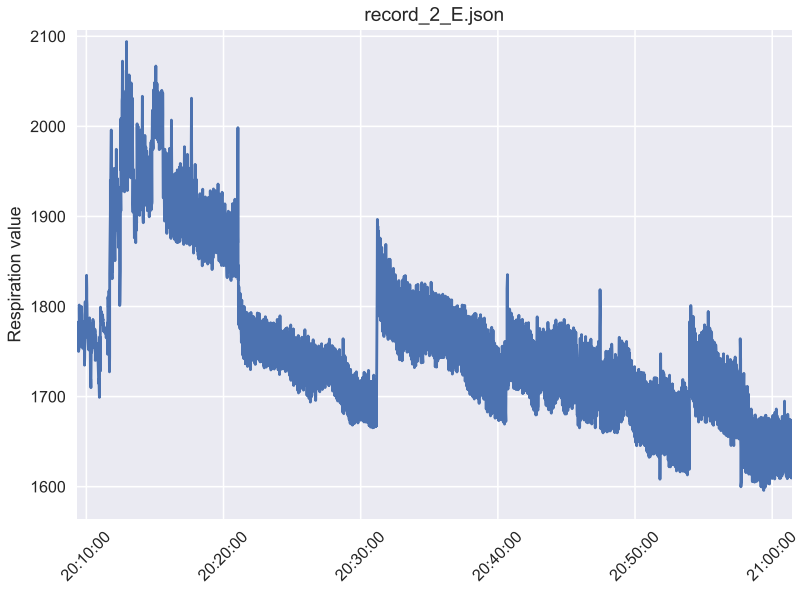

Figure B.11: Concert Day 2: Device E

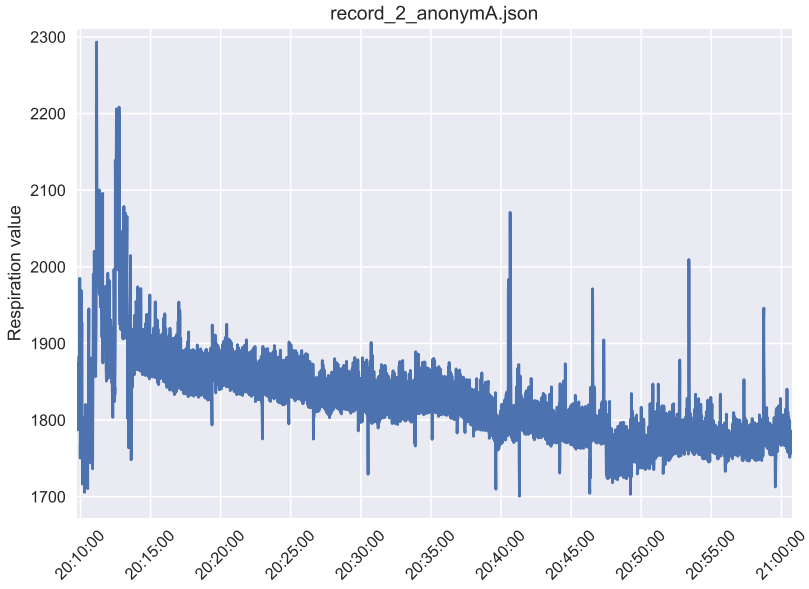

Figure B.12: Concert Day 2: Device F

B.3 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")
```

```
41      ax.set_title(name)
42
43      plt.show()
44
45      print(sample_count)
46
47  def read(filename="record_1_B.json"):
48      with open(filename) as f:
49          json_data = json.load(f)
50
51          parse(json_data, filename)
52
53  if __name__ == "__main__":
54      if len(sys.argv) == 2:
55          read(sys.argv[1])
56      else:
57          read()
```

Listing B.1: My Caption

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 latest Gradle version 3.4.0 and support for Android 9 (API level 28). In the subsequent Sections, the instructions of duplicating the template is 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 desired name, and move it to the `Android Studio` project folder. Listing X presents the commands to accomidate for the 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>
```

Listing C.1: My Caption

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>
```

Listing C.2: My Caption

C.1.3 Rename the Package of the Application

Change the package name of the application by following Listing X. It is incentivized to keep the prefix of *com.cesar.X* to group future modules by the package name.

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

Listing C.3: My Caption

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 Listing X.

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

Listing C.4: My Caption

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 seperated into 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 X)

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

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

As part of the discussion of the module design, the records is 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 temporarly cache the data under development.

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

[5] *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).

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

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

[10] Daniel Bugajski. 'Extensible data streams dispatching tool for Android'. In: (201).

[11] *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).

[12] 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).

[13] *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).

[14] 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).

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

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

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

[18] *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).

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

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

[21] 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).

[22] 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).

[23] Svein Petter Gjøby. 'Extensible data acquisition tool for Android'. In: (2016).

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

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

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

[27] 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).

[28] 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).

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

[30] 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).

[31] 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).

[32] 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).

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

[34] 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).

[35] 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).

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

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

[38] *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).

[39] *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).

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

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

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

[43] 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).

[44] 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).