Nidra

An Android application designed to record sleep with Flow sensors

Jagat Deep Singh

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

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Nidra

An Android application designed to record sleep with Flow sensors

Jagat Deep Singh

© 2019 Jagat Deep Singh

Nidra

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

Printed: Reprosentralen, University of Oslo

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Leo a diam sollicitudin tempor. Elit ullamcorper dignissim cras tincidunt lobortis feugiat vivamus. Consectetur a erat nam at lectus urna duis. Ullamcorper sit amet risus nullam eget felis eget nunc. Sollicitudin nibh sit amet commodo. Volutpat maecenas volutpat blandit aliquam etiam erat. Sed viverra ipsum nunc aliquet bibendum enim facilisis gravida neque. Metus dictum at tempor commodo. Nunc vel risus commodo viverra maecenas accumsan lacus. Vitae justo eget magna fermentum iaculis eu non diam. Habitant morbi tristique senectus et. Sed enim ut sem viverra aliquet. Lectus mauris ultrices eros in cursus.

Turpis massa tincidunt dui ut ornare lectus. Elit sed vulputate mi sit amet mauris commodo quis imperdiet. Etiam non quam lacus suspendisse faucibus interdum posuere lorem ipsum. Morbi non arcu risus quis. Quis viverra nibh cras pulvinar mattis nunc sed. Tellus cras adipiscing enim eu turpis egestas. Nec tincidunt praesent semper feugiat nibh sed. Ipsum dolor sit amet consectetur. Duis convallis convallis tellus id interdum. Nulla aliquet enim tortor at.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Leo a diam sollicitudin tempor. Elit ullamcorper dignissim cras tincidunt lobortis feugiat vivamus. Consectetur a erat nam at lectus urna duis. Ullamcorper sit amet risus nullam eget felis eget nunc. Sollicitudin nibh sit amet commodo. Volutpat maecenas volutpat blandit aliquam etiam erat. Sed viverra ipsum nunc aliquet bibendum enim facilisis gravida neque. Metus dictum at tempor commodo. Nunc vel risus commodo viverra maecenas accumsan lacus. Vitae justo eget magna fermentum iaculis eu non diam. Habitant morbi tristique senectus et. Sed enim ut sem viverra aliquet. Lectus mauris ultrices eros in cursus.

Turpis massa tincidunt dui ut ornare lectus. Elit sed vulputate mi sit amet mauris commodo quis imperdiet. Etiam non quam lacus suspendisse faucibus interdum posuere lorem ipsum. Morbi non arcu risus quis. Quis viverra nibh cras pulvinar mattis nunc sed. Tellus cras adipiscing enim eu turpis egestas. Nec tincidunt praesent semper feugiat nibh sed. Ipsum dolor sit amet consectetur. Duis convallis convallis tellus id interdum. Nulla aliquet enim tortor at.

Contents

List of Tables

List of Figures

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Background and Motivation

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, like Obstructive Sleep Apnea (OSA). Side effects of OSA do not only cause sleepiness during day time (which might affect daily chores), but also serious illnesses like diabetes and cardiac dysfunctions. Statistically speaking, it is estimated that about 25% of the adult population in Norway has OSA, but only 10% of them are diagnosed. A major problem with diagnosing OSA is polysomnography in *sleep laboratories* [3]. This is both really expensive and inefficient due to lacking capacity to perform sufficient tests with patients. Hence, the CESAR project aims to contribute to this situation with a low-cost Android and BiTalino based system to tackle these problems in a minimal invasive approach.

The project has been developed by various people over the years, and the system has been divided into three parts (illustrated in Figure 1.1). 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.

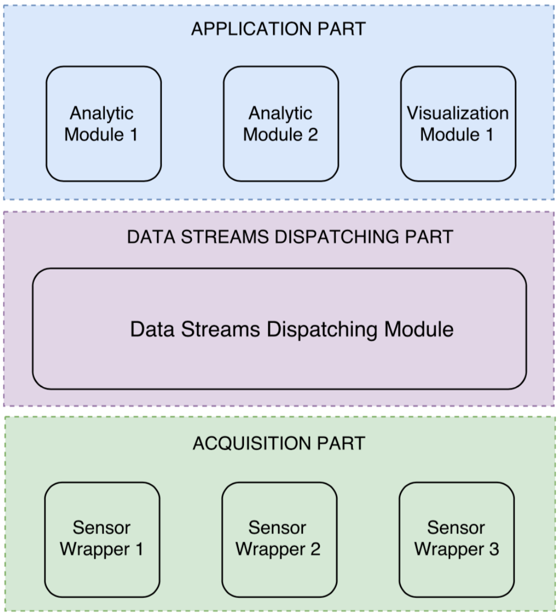

Figure 1.1: Structure of the project, separating functionality into three independent layers [1]

1.2 Problem Statement

As indicated in the background and motivation section, we decided to look into there are opportunity of extending the system 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. In the end, we can hopefully strengthen the detection of abnormal sleeping patterns, and decreasing the risk of the symptoms they may endure.

In this thesis, we will continue to build on the project by implementing the following:

1. *Integrate new a sensor wrapper*
   The following section Flow Sensor Kit introduces to a new sensor type to be used in the CESAR project. Thus, creating a new wrapper to integrate it in the system is necessary. By utilizing the library that comes with the sensor, we can integrate that with the interface and protocol of the CESAR project.

2. *Visualization of the activities on an Android device*
   Designing, architecting, and modulating an Android application that

implements the new sensor data in an adequately layout.

3. *Detecting anomalies with the help of Machine Learning*
   Classifying the data so we can detect abnormal sleeping patterns, and hopefully training an model that can detect the sleeping patterns, locally on the device, without any external supervisor (human intervention) analyzing the data.

1.3

Limitations

1.4 Research Methods

1.5 Contributions

1.6 Thesis Outline

Chapter 2

Background

2.1 Project Architecture

2.2 MVVM

2.3 Sleep Apnea

2.4 Flow Wrapper

2.5 Bluetooth LE

2.6 Android OS

2.7 Daniel

2.8 Viet

2.9 Svein

Chapter 3

Related Work

This chapter survyes previous work related to development of the CESAR project.

3.1 Extensible Data Acquisition Tool

In the thesis "Extensible data acquisition tool for Android" by Svein Petter Gjøby [2], 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

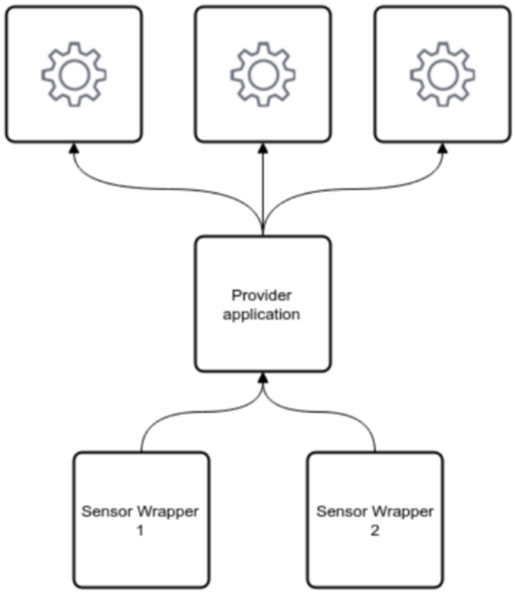

Figure 3.1: Sharing the collected data between multiple applications [2]

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.

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 3.1.

3.2 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 [1]. 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 can not 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 own 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 access from the application part at anytime.
2. *demultiplexer* (DMUX) - is a data cloner, that receives data packets from one input (i.e., from one channel), and duplicates the data a number of times - based on number of subscribers.
3. *publish-subscribe* mechanism - is an interface responsible for providing

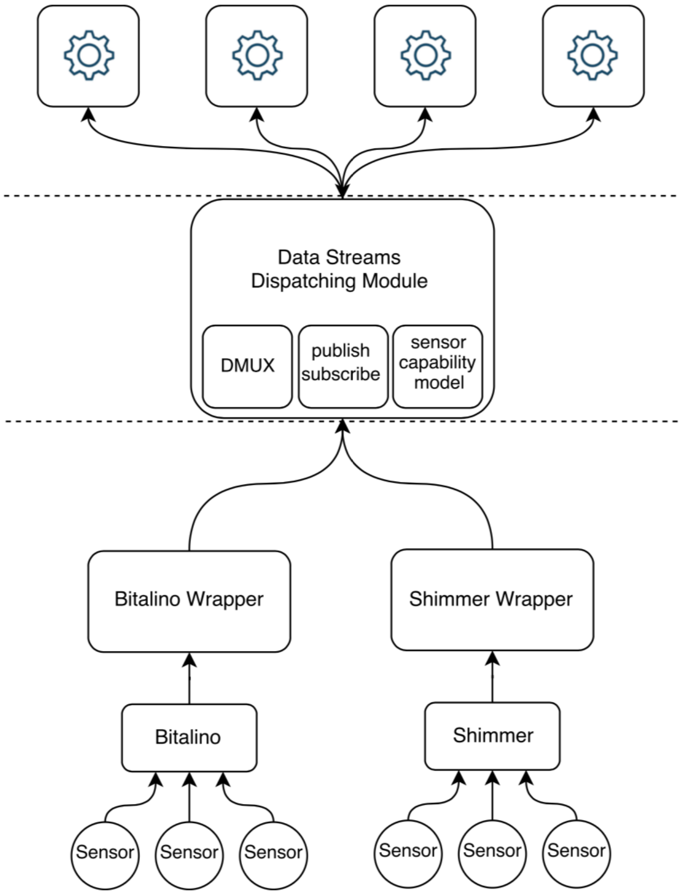

Figure 3.2: Sharing the collected data between multiple applications [1]

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, were 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 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 inserting the remaining elements in the respective application part. This solution resolves the overload of sensor wrappers performing other task 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 own sensor-capability model; (2) filter requested data packets from all the channels; (3) and deal with the collection speed on its own.

Addressing these issues leads us to the final architecture, which is presented in Figure 3.2, 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 very 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 of modifying large parts of the system. The system is also very effective, by the means of 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.

3.3 Database Model for Storing OSA Data

A database for storing Obstructive Sleep Apnea related data is discussed in the thesis "An open database model for storing Obstructive Sleep Apnea data" by Viet Thi Tran [4]. The thesis presents the design and implementation of a relational *database model* for **storing** OSA signals and bio-physiological signals, simplify the **analysis** of the signals, and supporting acquisition from **future data sources**.

In terms of storing data in a database system, the context of what the database is used for, and what it must contain, are usually crucial to identify the appliance of the database. For storing OSA related data, the database system must satisfying the requirements of the sensor sources, and requirements of the users of the system. The users in the system are *patients, physicians, and researchers*. A few characteristics of the actions the users can take are:

1. *Patients* – the users of the group are able to execute a simple function (i.e. inserting, deleting and queries). Such function might be finding records, storing sample from CESAR tool, and import/exporting certain recordings. Mostly, their action are to import and to export.

2. *Physicians* – are able to apply functions that modify existing data, manually training data for a recording, retrieve all recording of patients, etc... In other words, they have knowledge of OSA health data and may tweak values based on their expertise in the field.

3. *Researchers* – often evaluates tasks on the system to find the best solution. Thus, they would most likely want to evaluate the quality of the source that is used for collecting the signals. Some actions they can take are to evaluate quality of sources and channels, perform raw query to find a cost and performance beneficial query, and applying possible mining algorithms.

The benefits of using a relational database to store the OSA related signals are: *data analysis* can be performed on client's device (e.g. mobile devices) by utilizing SQL and its supporting algorithms; *remote services* to fetch parts of the client's data by using remote querying; and *privacy of patients* is not violated as the data remains on their devices and they can decide which data they would like to share.

A logic data model describes the abstract structure of the database system without considering physical structure, i.e., how the database is implemented. The features of a logical data model includes: *tables (entities)* and all relations between them; *columns (attributes)* for the entities; *primary key* for all of the entities; and *foregin key* for identifying relations between different entities.

Before analyzing the proposed data model of the system, we identify the entities that are part of the system:

- **Source** – is the entity storing the data obtained from CESAR acquisition tool, in addition to EDF files. Mainly, the entity stores the bio-signals data from sensor sources (such as BiTalino).

- **Recording** – is a session of sample recorded on the *users* device.

- **Person** – is both the *Patient* and the *Physician* (however, I could not find any listing of *Researchers* in the thesis), because these two entities share common attributes.

- **Clinic** – provides advanced diagnostic or treatment services for specific diseases. The bio-physiological signals are usually stored in formats tailored specific to a clinic. Thus, the clinics have their own way of formatting and manipulating the bio-physical signals to their standards, and the data may be varying between clinics.

The actual data model, we can define the relationship between the entities based on the identified requirements. In Figure 3.3, we have a logical data model for the entities in the system, based on the following relations:

a) Each Source has many Recordings, but one Recording belongs to only one Source.

b) Each Source can be used by many different Persons, and each Person can use many different Sources.

c) Each Source can be used by many different Clinics, and each Clinic can use many difference Sources.

d) Person has many Recordings, but one Recording belongs to only one Person.

e) Person collects many Recording, but one Recording is collected by only one Person.

f) Each Recording is produced by a Source, for a Person at a certain Clinic at a certain time.

g) Each Person works/belongs to many Clinics, and each Clinic employs/has many Persons.

h) Each Person (Physician) observe many Persons (Patient), and many Person (Patient) are observed by a Persons (Physician).

Normalization is a technique in relational database used for integrity, maintainability and performance of database. The purpose is to reduce the redundancy of the data stored in a database system, so the database can become more reliable and efficient. The table of data can be classified on: (1NF) first normal form; (2NF) second normal form; (3NF) third normal form; (BCNF) Boyce-Codd normal form; and (4NF) fourth normal form. Where a higher degree of a normal form is preferable.

In the thesis, the entities (source, recording, person, and clinic) of the system have their normal form evaluated by determining the functional dependencies and the multivalued dependencies based on the attributes of the entity. The normal form can then be derived by applying set of algorithms and rules (i.e., determining the normal form of the FDs/MVDs and decomposing it until the tables is on BCNF/4NF). By doing this, we ensure the data to be loss less on joins, in addition to reducing the redundant storage. The end result of the following steps, splits some of the entities into smaller groups of entities with common attributes:

The final result of the logical data model is presented in Figure 3.4 and 3.5. Theoretically, the model can be implemented and deployed on a database management system, which then can be used to store Obstructive Sleep Apnea data.

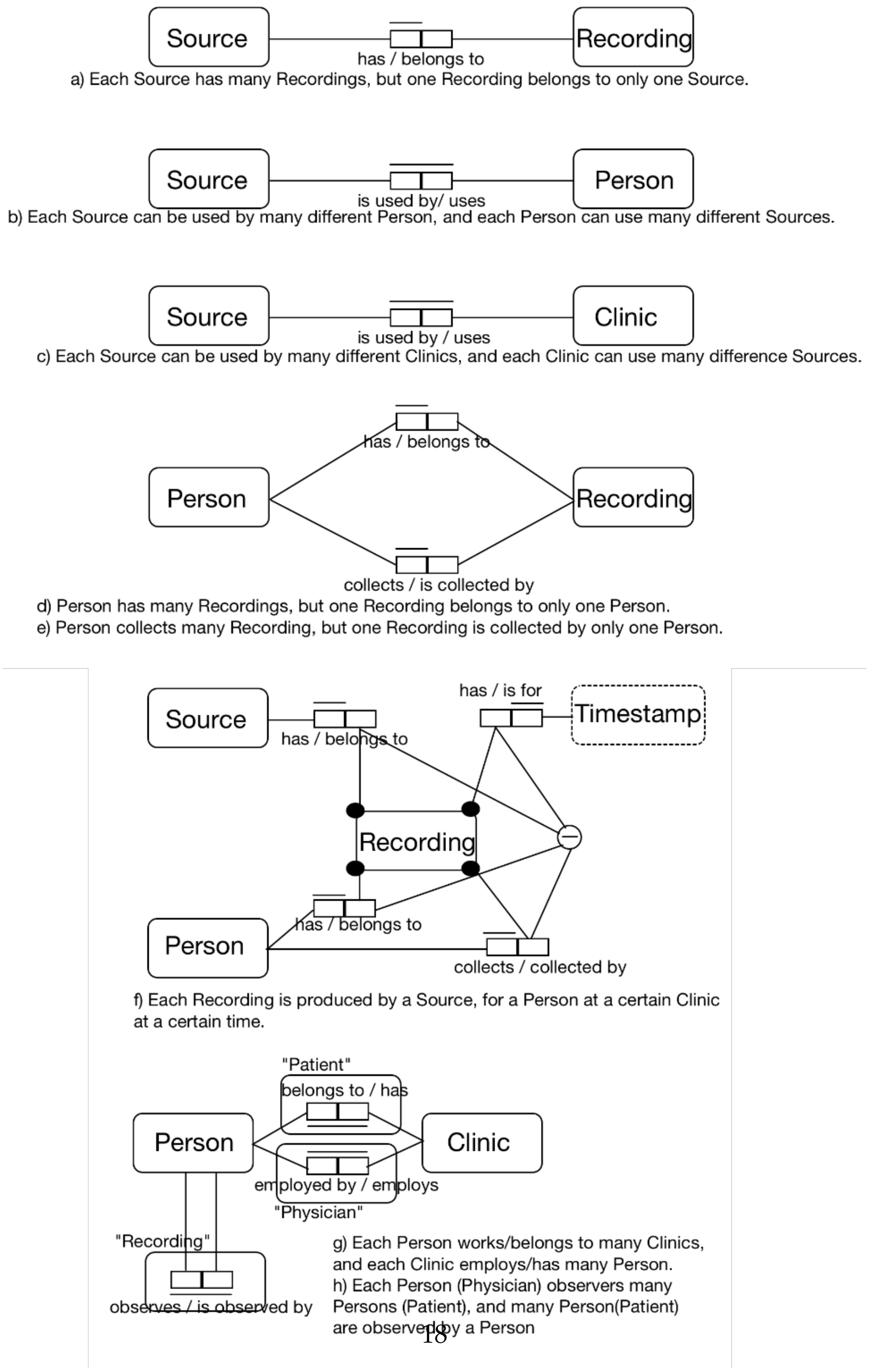

Figure 3.3: Binary, recursive and n-ary relationships [4]

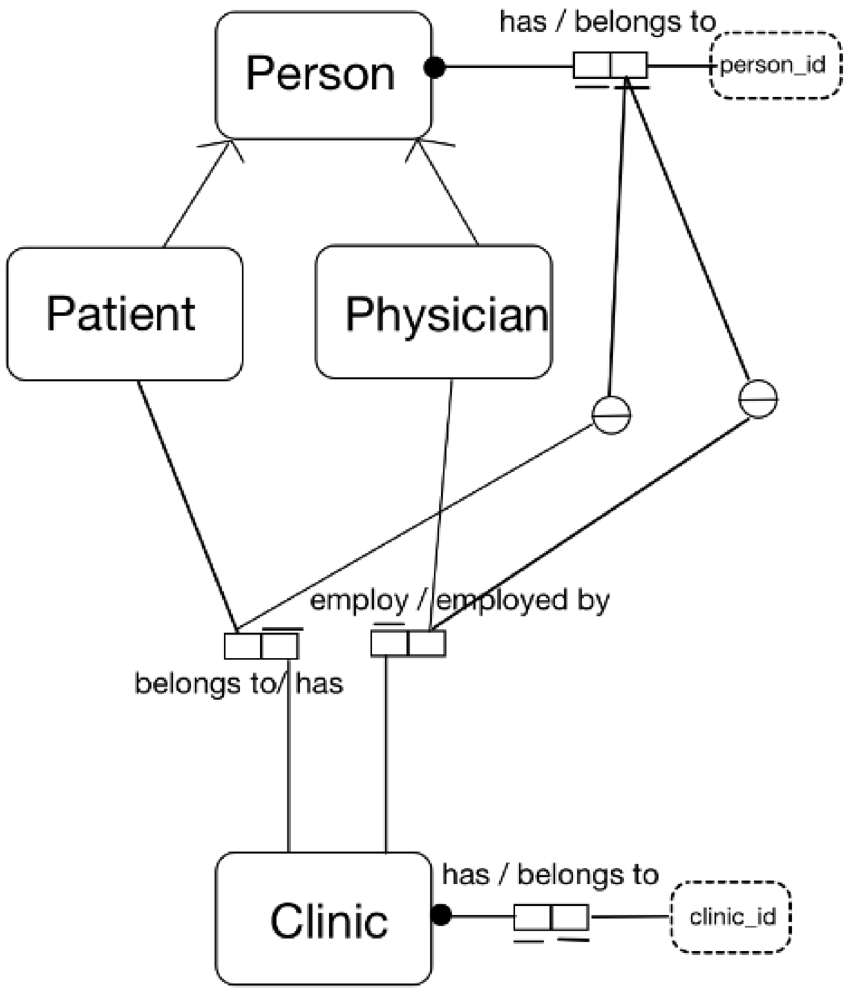

Figure 3.4: Logical model of the OSA database - Person and Clinic [4]

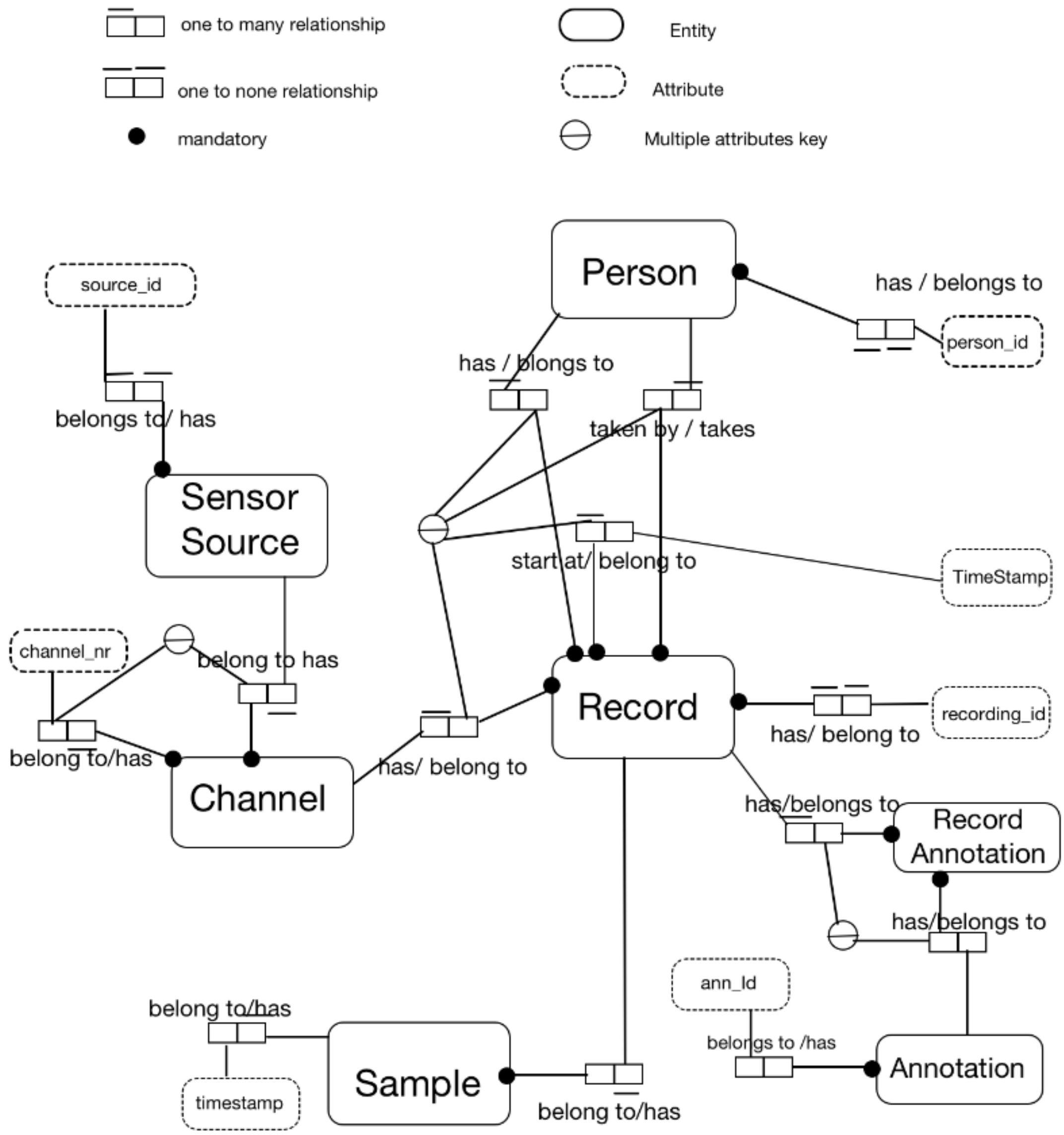

Figure 3.5: Logical model of the OSA database - Source and Recording [4]

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 (of the illness?) 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 (i.e., during sleep); 2) present the results; and 3) export/import 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...

4.1 High-Level Design

4.1.1 Stakeholders

A stakeholder is a term coined to describe those persons or organizations that have, or claim an interest in the project [Stakeholder defined, p. 4]. Identifying stakeholders is essential to fulfilling the requirement set in the thesis, as they contribute to form and sculpture the application. From the article [stakeholder defined, p.14] we can distinguish stakeholders into four categories: 1) *contributing (primary) stakeholders* are those that participate in developing and sustaining the project; 2) *observer (secondary) stakeholder* are those who affect or influence the project; 3) *end-user (tertiary stakeholder)* is the one who interact and uses the output of the application; and 4) *invested stakeholder* is one who has control of the project [stakeholder defined, p. 13]. In Nidra, we have 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. Additionally, 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 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 [Task analysis...]. In this Section, we will be analyzing system tasks and user-related tasks.

4.1.2.1 System Tasks

Recording

A *recording* is a process of collecting and storing physiological signals from sensors over an extended period (i.e., overnight). To enable a recording, we need to establish connections to available sensors, collecting 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. physiological signals over a medium (e.g., Bluetooth). The application and the sensors are separate components, and the communication between these components occurs over an application programming interface (API) [write more?]. A *sample* is a single sensor reading containing data and metadata, such as time and physiological data.

Sharing

Sharing is a mechanism to export and import recordings across applications. *Exporting* consists of bundling records with correlated samples into a format that can be transmitted. The format can be structured varyingly, however, we will discuss as few distinguishable formats that are applicable:

- JSON - is a file format used to transmit data objects consisting of attribute-value pairs and array data types [wikipedia, JSON, 9.mai]. JSON has a simple syntax, which results in a compact file and efficient transmission. However, it only supports a few data types.
- XML - is a markup language that encodes arbitrary data structures into a format that is human-readable and machine-readable [wikipedia, XML, 9.mai]. XML provides a generalized markup that has support for numerous data types, structure validation, and extensions. However, the structure of XML results in a larger file[?]
- Constructing a file format solely for transmitting records and samples - by introducing a file format that is restrictive to the purpose of records and sample, we can minimize the transmitting file size. However, this might result in unreadable text, the overhead of parsing and transforming, and incompatibilities amongst devices.

While these format structures are applicable for transmitting data, choosing a format that is compact, human-readable, and universal is essential. In order to design this application, we will be using the JSON format for transmission of the data, as it meets the specified criteria.

Once a preferred format for the recording is selected and stored in an exportable file, transferring the file to another device is made possible. One way is by transferring the files with the desired recipient through a server/middleware[?]; this makes it easier for users to share files amongst each other. Another way of transferring a file from one device to another is by sharing it through applicable mediums (e.g., email). The former requires additional functionality, such as registering users to distinguish recipients and additional security measurements for securing personal information. Therefore, a more preferable and secure solution would be to use the former solution.

Importing is accomplishable by locating the file, parsing the file, and storing the recordings in storage. A naive solution for the location of a file is by assuming that the file is located in on the same location amongst all devices, thus trying locating the file on a static location. Therefore, providing an interface to the users to deliberately locate the desired file in the file hierarchy of its device is practical. Android provides an interface for such a solution [?]. With the exact path of the file, we can read the bitstream of the file and parse the data according to the chosen format, and store the content of the file on the device.

Modules

Modules are independent applications that can be launched in the application to provide extended functionality and data enrichment. A module does not necessarily interact with the application itself. However, it utilizes the data (e.g., recordings) provided from the application. For instance, a module could be using the recordings to feed a machine learning algorithm to predict obstructive sleep apnea. Installing a module can be done by locating the module-application on the device, and can then be launched from the application. However, due to limitations in Android, the module-application cannot be executed within the application. Therefore, modules are run as independent application alongside the application.

The data exchange between a module and the application is possible on two various methods. One way is by selecting one or all of the recordings and bundling the data and sending it on launch. Another way is by establishing a direct communication link for pull-based requests, where the records are sent based on the requests of the module. The latter solution provides less overhead on data transfer; however, requires extended functionality to made possible. The former solution sends all of the selected data to the module on the launch, and there are no methods of communication with the application once a module is running. Essentially, the recordings do not change once it is on the device. Thus the former solution is feasible.

Storage

Storage is the objective of achieving persistent data; data that is available after application termination. To enable storage, we can use a database for a collection of related data that is easily accessed, managed, and updated [Database Systems, p. 52]. Three distinguishable databases structures are:

- Flat file - encode a database model (e.g., table) with a collection of records without any structured relationship, in a plain text or binary file [system nosql analysis].
- Relational Database - consists of relations between data stored in tables; supporting complex queries, database transactions, and Additionally, ensuring ACID (atomicity, consistency, isolation, and durability) properties for reliable database transactions.
- Non-Relational Database - While these database structures are applicable to achieve persistent data storage, using relational database is suitable for our

The placement of storage can be located externally on a server or internally on the device. External storage provides larger storage capacity, in addition to faster computing power. Facilitating external storage can be accomplished with maintaining a server, or utilizing a cloud service (e.g., Google Cloud) for the purpose. Internal storage provides limited storage capacity and computing power, however, the storage and retrieval data is efficient. While both are applicable, the internal storage is more than efficient for the application.

Two distinguishable methods of structuring a storage There are several methods of structuring the storage on, and two distinguishable methods are a flat file or a database management system (hereafter, DBMS). File storage is a bare minimum structure of storing and retrieving data. The data is stored, with a format (e.g., JSON), in a raw file. Retrieving data occurs by reading the whole file into memory, parsing the data and then operating on the data.

4.2 Seperation of Concerns

4.2.1 Recording

There are several approaches to assemble components to achieve a recording, and we review an alternative structure. In Figure 4.1 we have an HTA graph, which illustrate the building blocks to enable a recording and their dependencies:

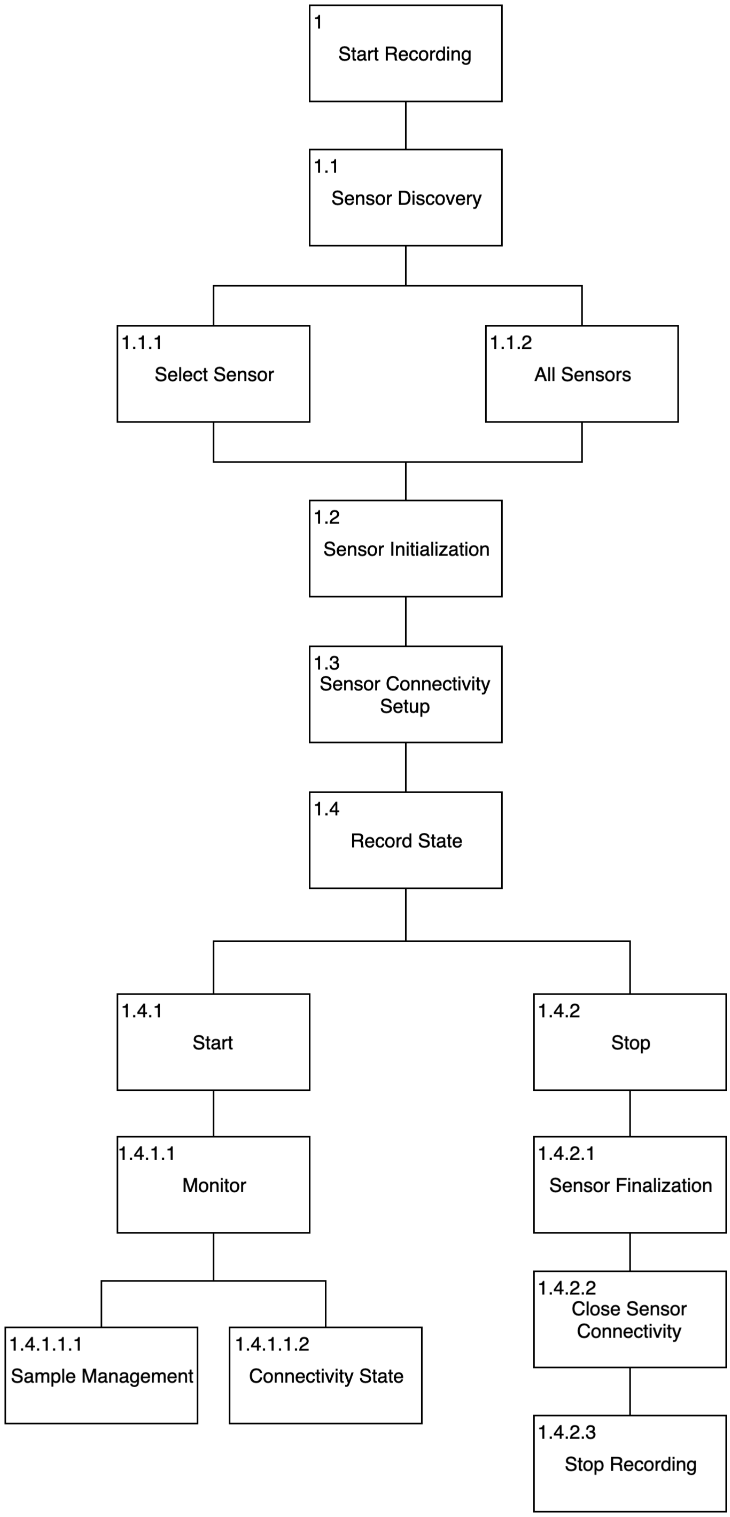

Figure 4.1: Recording

- Sensor Discover: Has to find all eligible sensors that can enable a recording.
- Select Sensors: From the sensor discovery, we can choose preferable sensors sources.
- All Sensors: More straightforward, we sample from all of the available sensors.
- 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. Thus, blocking the state of the recording.
- Sensor Connectivity Setup: Additionally, we establish a connection between the application and the sensor source. All data exchange occurs over the established interface.
- Connection Stat: Based on sensors establishments we can proceed to either start or stop a recording.
- Start: By starting, we notify the sensors to begin collecting data, and the view should display that a recording has begun accordingly.
- Monitor: Is continuously waiting for new samples to arrive on the interface defined between the application and the sensors.
- Sample Management: Once a new sample has arrived, we need to store the sample on a persistent storage.
- Connectivity State: If it is an external sensor, the sensor source might disconnect during a recording. Thus, implementing a mechanism to check for continuous data stream is a critical task.
- Stop: By stopping, we notify the sensors to stop collecting data from the sensor source.
- Sensor Finalization: We notify the sensor to stop sampling data, and close establishment.
- Close Sensor Connectivity: We close the interface establishment between the application and the sensors.
- Stop Recording: Once the sensors has closed its connections, we can add additional information to the recording (e.g., title, description, rating). In the end, the recording has concluded and its stored on the mobile device.

This suggested structure of a recording is one alternative to enable a recording. Most of the components suggested in the structure are essential to a recording. A naive solution would be to ignore the connectivity state component, by assuming the sensors are connected indefinitely. In our thesis, we will be following

4.2.2 Sharing

4.2.3 Modules

4.2.4 Storage

4.2.5 Presentation

Chapter 5

Implementation

Part III

Evaluation and Conclusion

Chapter 6

Experiments

Chapter 7

Conclusion

Appendix

Appendix A

Power Data

Bibliography

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

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

[3] Stein Kristiansen Thomas Peter Plagemann Vera Hermine Goebel. *Android based eHealth applications with BiTalino sensors*. URL: http://www.mn.uio.no/ifi/studier/master based-ehealth-applications-with-bitalino-s.html (visited on 05/05/2018). 2015. URL: http://www.mn.uio.no/ifi/studier/masteroppgaver/dmms/android-based-ehealth-applications-with-bitalino-s.html.

[4] Viet Thi Tran. "An open database model for storing Obstructie Sleep Apnea data". In: (2017).