

Nidra

*An Android application designed to
record sleep with Flow sensors*

Jagat Deep Singh



Thesis submitted for the degree of
Master in Network and system administration
30 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 Listings	vi
List of Tables	vii
List of Figures	ix
I Introduction and Background	1
1 Introduction	3
1.1 Background and Motivation	3
1.2 Problem Statement	3
1.3	4
1.4 Research Methods	5
1.5 Contributions	5
1.6 Thesis Outline	5
2 Background	7
2.1 Project Architecture	7
2.2 MVVM	7
2.3 Sleep Apnea	7
2.4 Flow Wrapper	7
2.5 Bluetooth LE	7
2.6 Android OS	7
2.7 Daniel	7
2.8 Viet	7
2.9 Svein	7
3 Related Work	9
3.1 Extensible Data Acquisition Tool	9
3.2 Extensible Data Stream Dispatching Tool	11
3.3 Database Model for Storing OSA Data	14
II Design and Implementation	21
4 Analysis and High-Level Design	23
4.1 Requirement Analysis	24

4.1.1	Stakeholders	24
4.1.2	Resource Efficiency	24
4.1.3	Security and Privacy	24
4.2	High-Level Design	25
4.2.1	Task Analysis	25
4.3	Seperation of Concerns	27
4.3.1	Recording	27
4.3.2	Sharing	30
4.3.3	Modules	32
4.3.4	Analytics	33
4.3.5	Storage	34
4.3.6	Presentation	35
4.4	Data Structure	35
4.4.1	Data Formats	35
4.4.2	Data Packets	37
5	Implementation	39
III	Evaluation and Conclusion	41
6	Experiments	43
7	Future Work	45
8	Conclusion	47
	Appendix	48
A	Power Data	51

Listings

4.1	My Caption	36
4.2	My Caption	36
4.3	My Caption	37

List of Tables

List of Figures

1.1	Structure of the project, separating functionality into three independent layers [3]	4
3.1	Sharing the collected data between multiple applications [6]	11
3.2	Sharing the collected data between multiple applications [3]	13
3.3	Binary, recursive and n-ary relationships [10]	17
3.4	Logical model of the OSA database - Person and Clinic [10] .	18
3.5	Logical model of the OSA database - Source and Recording [10]	19
4.1	Recording	25
4.2	Recording	28
4.3	Sharing	31
4.4	Modules	32
4.5	Analytics	33
4.6	Storage	34
4.7	Presentation	35

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* [9]. 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.

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

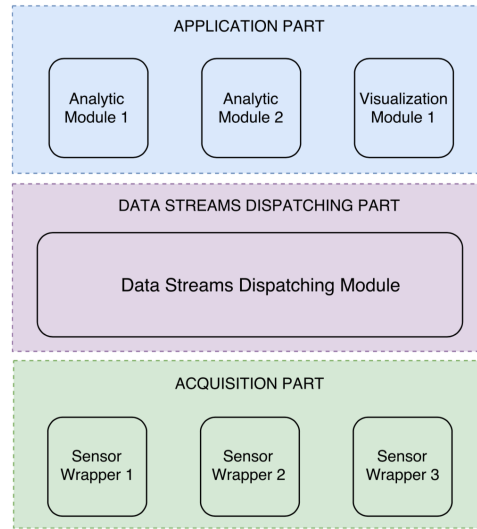


Figure 1.1: Structure of the project, separating functionality into three independent layers [3]

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 surveys 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 Gjølby [6], 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 independent 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 functionalities outweighs 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

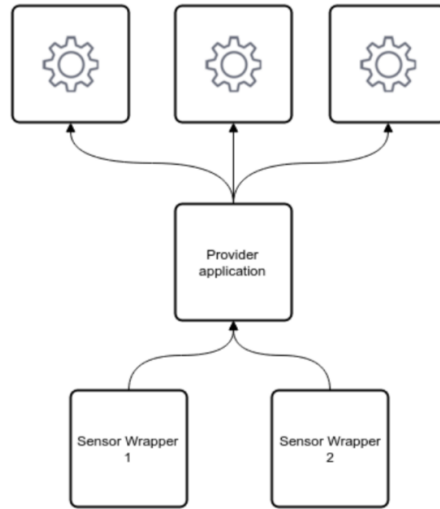


Figure 3.1: Sharing the collected data between multiple applications [6]

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 one 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 [3]. 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 possibilities 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.

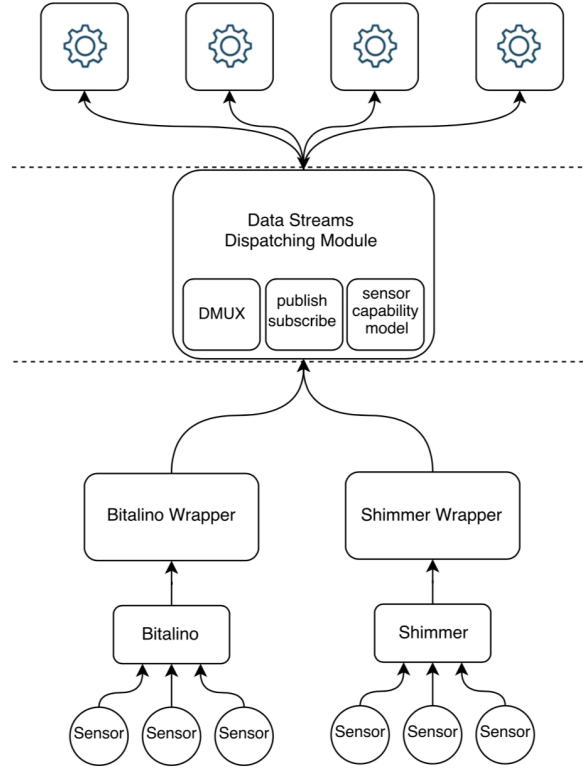


Figure 3.2: Sharing the collected data between multiple applications [3]

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 model. 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 [10]. 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 *foreign 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.

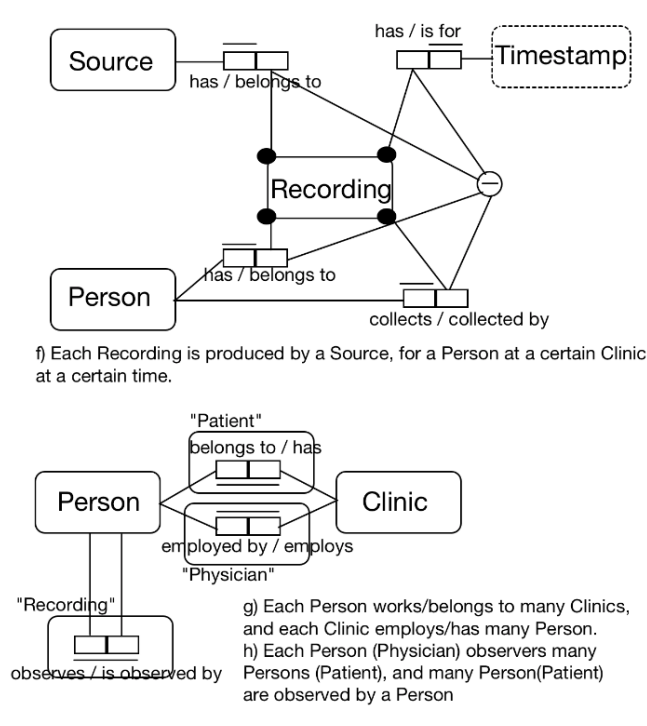
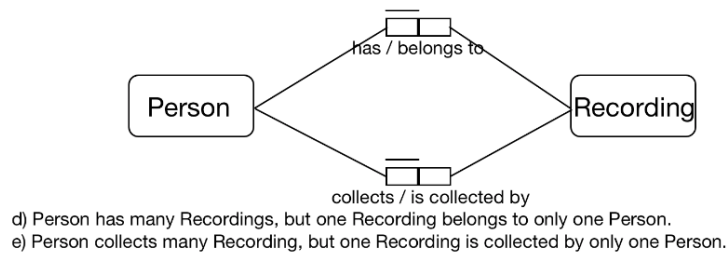
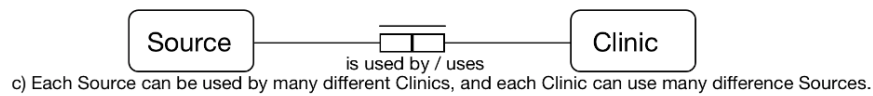
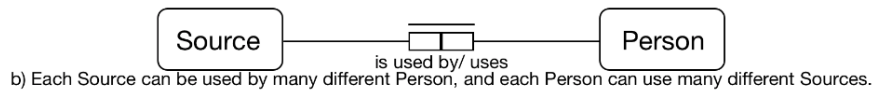
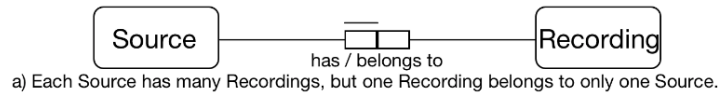


Figure 3.3: Binary, recursive and n-ary relationships [10]

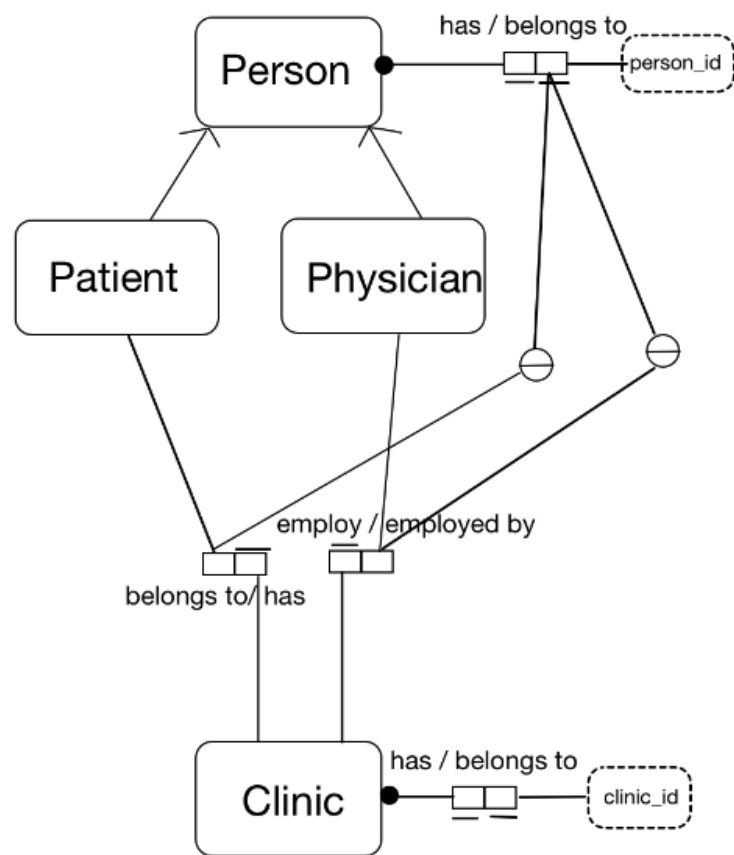


Figure 3.4: Logical model of the OSA database - Person and Clinic [10]

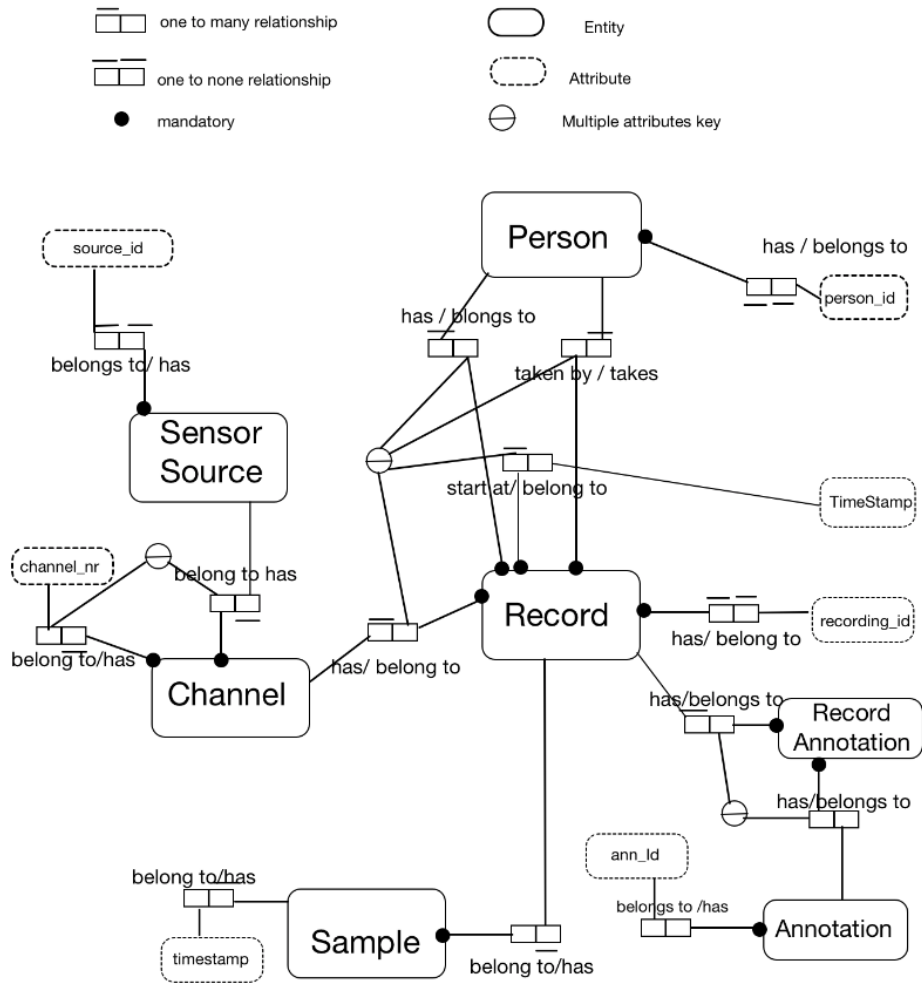


Figure 3.5: Logical model of the OSA database - Source and Recording [10]

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 Requirement Analysis

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. Identifying stakeholders is essential to fulfilling the requirement set in the thesis, as they contribute to form and sculpture the application. As discussed by McGrath *et al.* [8], stakeholders can be distinguish 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. 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 functionality of the application. Additionally, they can contribute to developing modules that extend the functionality of the application.

4.1.2 Resource Efficiency

4.1.3 Security and Privacy

The proposed use of the application is to monitor the sleeping patterns of a user. The application manages and stores personal-data and health-related data about the user on the device. As a consequence, the application should incorporate the CIA triad, which stresses data confidentiality, integrity, and availability [11]. Any unauthorized access to the data, data leaks, and confidentiality should be appropriately managed on the device. Sharing the data across application or with other users should be granted with the consent from the user. Besides, a mobile device can be connected to the Internet, which makes it vulnerable to attacks. Also, installed application on the device can manipulate the access to the data. Therefore, revising the security policy defined by Android [2] should be contemplated.

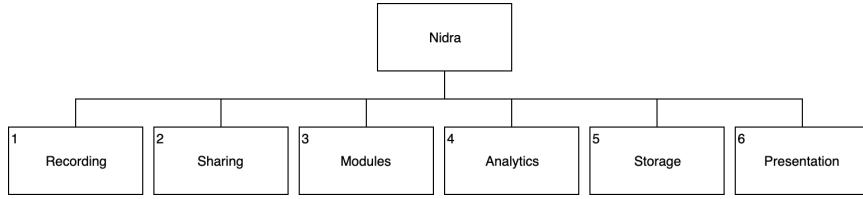


Figure 4.1: Recording

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 [5]. In Figure 4.1 an illustration of the the main components of the project is presented. These components are integral to the development of the project, and in this Section, we will be analyzing system tasks and user-related 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. The digital signals are transmittable over a medium (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 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 recordings across applications. *Exporting* consists of bundling record with correlated samples into a transmittable format, and transferring the bundled recordings over a medium (e.g., mail). *Importing*, on the other hand, consists of locating the bundled record on the device, parsing the content and storing it on the device.

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., recordings) provided by the application. For example, a module could be using the records provided by the application 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 to the module in the application. The module-application is a standalone Android application; thus, the development of the module independent from the application. Due to limitations in Android, the module-application cannot be executed within the application. Hence, modules are running alongside the application.

Analytics

Analytics is the discovery and interpretation of meaningful patterns in data [wikipedia, analytics]. 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 regression to advanced machine learning. However, 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, we provide a graphical representation of the data that can be further analyzed within the application.

Storage

Storage is the objective of achieving persistent data; data that is available after application termination. To enable storage, we use a database for a collection of related data that is easily accessed, managed, and updated [Database Systems, p. 52]. The database should be able to store records, samples, modules, and users (/ storing biometrical data related to user) adequately. Thus, structuring a database that is reliable, efficient, and secure is a crucial part of achieving persistent storage. Additionally, determining the database location, whether the database should be located on the device or located on an external server, is essential in the design of the application.

A *user* is a person that interacts with the application and its functionalities. While a user is not directly correlated to a task in the application, identifying its cognitive load and interactions helps on defining the features and structure of the application. A user has the possibility of providing biometrical data (i.e., gender, age, height, and weight), which can be used to enrich a recording to strengthen the outcome potentially.

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 between views, and feedback on actions is part of the development of a user interface.

4.3 Separation 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 [7]. 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.

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 (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:** Has to 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. Thus, blocking the state of the recording.
- 1.3 **Sensor Connectivity Setup:** Additionally, we establish a connection between the application and the sensor source. All data exchange occurs over the established interface.
- 1.4 **Connection Stat:** Based on sensors establishments we can proceed to either start or stop a recording.
 - 1.4.1 **Start:** By starting, we notify the sensors to begin collecting data, and the view should display that a recording has begun accordingly.

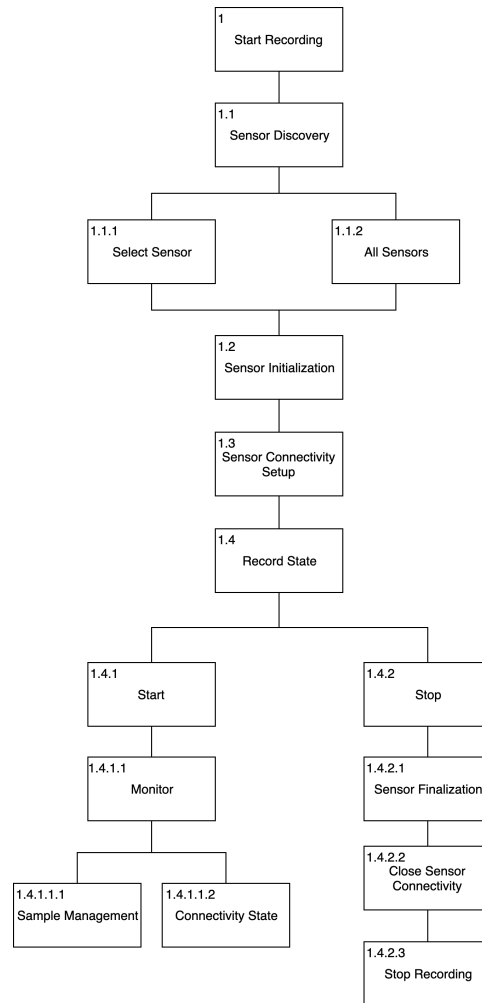


Figure 4.2: Recording

- 1.4.1.1 Monitor: Is continuously waiting for new samples to arrive on the interface defined between the application and the sensors.
- 1.4.1.1.1 Sample Management: Once a new sample has arrived, we need to store the sample on a persistent storage.
- 1.4.1.1.2 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.
- 1.4.2 Stop: By stopping, we notify the sensors to stop collecting data from the sensor source.
- 1.4.2.1 Sensor Finalization: We notify the sensor to stop sampling data, and close establishment.
- 1.4.2.2 Close Sensor Connectivity: We close the interface establishment between the application and the sensors.
- 1.4.2.3 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.

4.3.1.1 Connectivity State Component

Connectivity state is a component that monitors for unexpected sensor disconnections or abruptions. 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 abruptions, 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, 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 abruptions, 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. Though, by extending the functionality of sensor wrapper by implementing a state that indicates whether a recording is undergoing or stopped solves the prob-

lem. 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 abrupted. 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 unnecessary deterioration 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 dispatching the bundle over a medium (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:

2.1.1 Locate File: locate the file on the device by using Android picker.

2.1.2 Parse Formated Content: parse the content of the file

2.1.3 Insert Storage: with the parsed data, insert it into the storage

2.2 Export:

2.2.1.1 All Records: export all of the records on the device

2.2.1.2 Select Record: choose one record to export

2.2.2 Format File: When a preferred format for the records is selected, bundling the data into a file for transmittal over a medium can be done. It is essential to identify the name of the file uniquely to prevent duplicates and overrides of data. For instance, it is identifying the

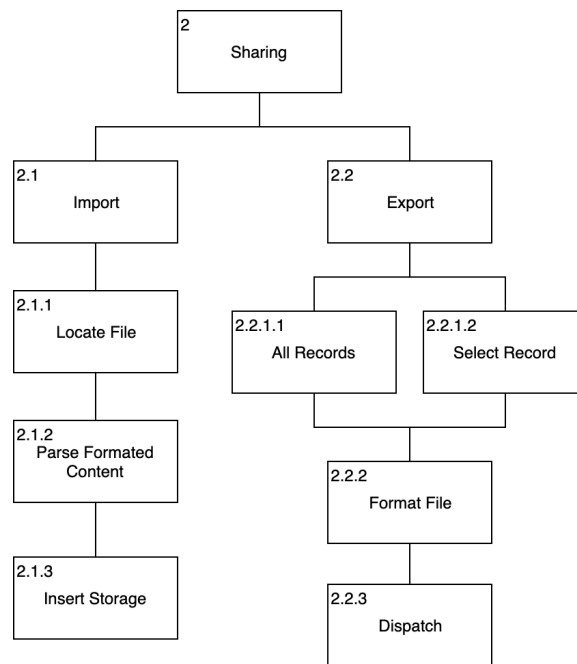


Figure 4.3: Sharing

name of the file with the device identification appended with the time of exporting.

2.2.3 Dispatch: send the file over a medium (e.g., mail).

4.3.2.1 Dispatching Component

The dispatching component uses the formatted file and transfers it across applications. There are two distinctive methods to perform named 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 medium to transfer the file, and the user can specify the recipients accordingly.

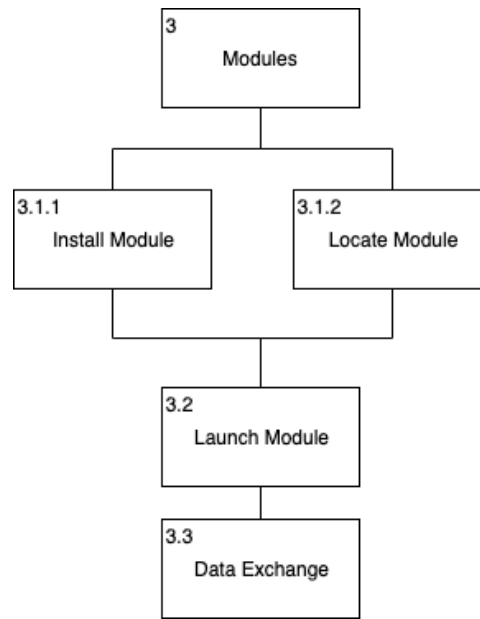


Figure 4.4: Modules

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:

3.1.2 Locate Module: locate the file on the device.

3.2 Launch Module: parse the content of the file

3.3 Data Exchange: with the parsed data, insert it into the storage

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 pull-based requests to Nidra from 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

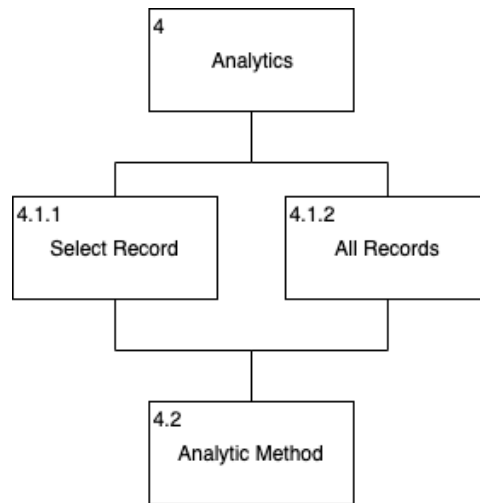


Figure 4.5: Analytics

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; thus, 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 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.

4.1.2 All Records: parse the content of the file

4.2 Analytic Method:

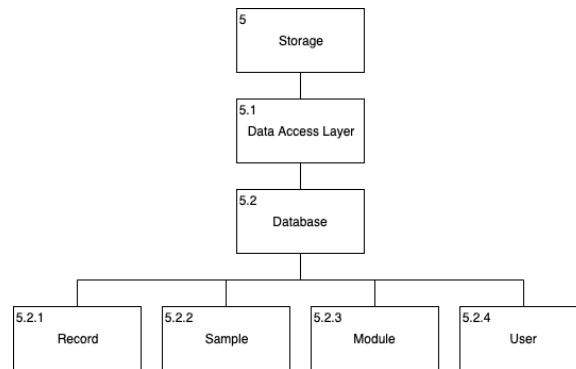


Figure 4.6: Storage

4.3.5 Storage

Storage is the objective of achieving persistent data; data that is available after application termination. To enable storage, we can use a database that stores data for easy access, manage and update. Extending a database with a data access layer allows for decoupling and extensibility, in case of future data modification and change. The data in Nidra revolves around modules, records, samples, and user. In Figure 4.6, the structure of storage is presented with components and their dependencies:

5.1 Data Access Layer:

5.2 Database:

5.2.1 Record:

5.2.2 Sample:

5.2.3 Module:

5.2.4 User:

4.3.5.1 Database

Choosing a database depends on specific criteria defined for the application data. Three distinguishable database structure 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

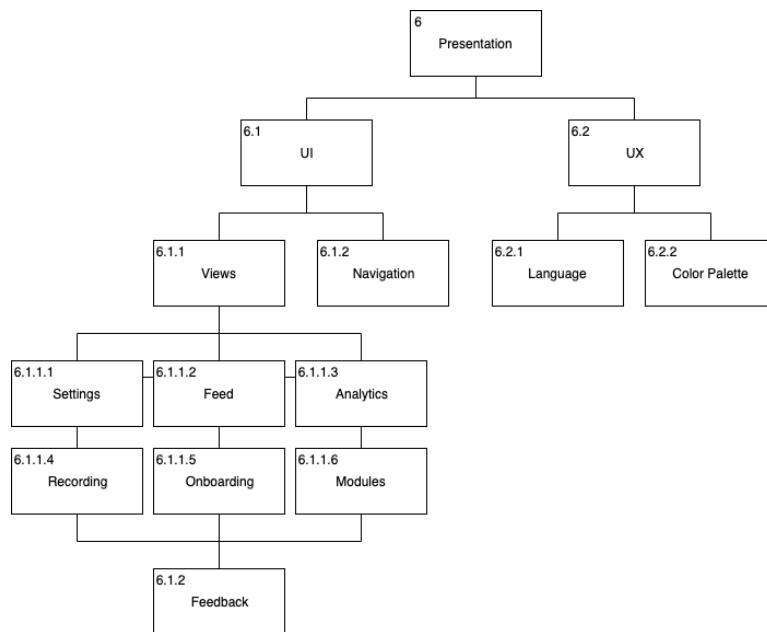


Figure 4.7: Presentation

suitable for our test

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.

4.3.6 Presentation

The format file component converts selected records into a formatted file. Formatting is used to serialize the data to enable transmission over the internet. 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, and a few appropriate formats are:

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 these formats and chose a 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, 12]. However, it is limited to a few predefined data types (i.e., string, number, boolean, object, array, and null), and extending the data type has to be done with the preliminary types.

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

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). [4, 12] XML provides mechanisms to define custom data types, using existing data types as a starting point, making it extensible for future data.

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

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 [12]. 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. XML is document oriented, making it challenging to map objects in the OOP paradigm, while JSON is data oriented and is closer to the graph structure of objects in the OOP paradigm.

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 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.

4.4.2.1 Sharing

In Section (ref), 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.

```
1 {  
2   "record":{  
3     "id": 1,
```

```

4      "name": "Record 1",
5      "rating": 2.5,
6      "description": "",
7      "nrSamples": 6107,
8      "monitorTime": 5963088,
9      "createdAt": "Apr 4, 2019 9:30:56 PM",
10     "updatedAt": "Apr 4, 2019 9:30:56 PM",
11     "user": {
12         "age": 50,
13         "createdAt": "---",
14         "gender": "Male",
15         "height": 180,
16         "name": "Ola Nordmann",
17         "weight": 60
18     }
19 },
20 "samples": [
21     {
22         "explicitTS": "Apr 4, 2019 5:51:26 PM",
23         "implicitTS": "Apr 4, 2019 7:51:26 PM",
24         "recordId": 1,
25         "sample": "Time=0ms, deltaT=100, data
                    =1906,1891,1884,1881,1876,1718,1690"
26     },
27     ...
28 ]
29 }

```

Listing 4.3: My Caption

4.4.2.2 Sensor Data

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

Chapter 5

Implementation

Part III

Evaluation and Conclusion

Chapter 6

Experiments

Chapter 7

Future Work

Chapter 8

Conclusion

Appendix

Appendix A

Power Data

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] Android. *Secure an Android Device*. URL: <https://source.android.com/security> (visited on 30/05/2019).
- [3] Daniel Bugajski. 'Extensible data streams dispatching tool for Android'. In: (201).
- [4] 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).
- [5] 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).
- [6] Svein Petter Gjøby. 'Extensible data acquisition tool for Android'. In: (2016).
- [7] 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).
- [8] 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).
- [9] Stein Kristiansen Thomas Peter Plagemann Vera Hermine Goebel. *Android based eHealth applications with BiTalino sensors*. URL: <http://www.mn.uio.no/ifi/studier/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>.
- [10] Viet Thi Tran. 'An open database model for storing Obstructive Sleep Apnea data'. In: (2017).

- [11] 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).
- [12] 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).