Exploring Fitbit smartwatches to detect sleep related disorders

Adem Salih

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

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Exploring Fitbit smartwatches to detect sleep related disorders

Adem Salih

Exploring Fitbit smartwatches to detect sleep related disorders

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

Printed: Reprosentralen, University of Oslo

Abstract

In recent years, smartwatches have gained increased attention due to their compact size and various built-in sensors. Conventionally, the sensors are used for evaluating physical activity and health, but may also be used to detect sleep-related disorders such as obstructive sleep apnea (OSA). OSA is an underdiagnosed sleep disorder where the upper airways are blocked for periods during sleep [1]. As a result, the level of oxygen in the blood decreases. Certain smartwatches with pulse oximeter sensors may be used for detecting apnea events, but the accuracy of such sensors must be assessed thoroughly by collecting data from the sensor.

In this thesis, we design and implement a smartwatch application, Hypnos, for acquiring sensor data from built-in sensors of Fitbit smartwatches. Hypnos is able to collect sensor data from the hardware sensors at various frequencies and reliably transfer the data to our smartphone application, Nyx. In addition, we investigate the available tools and technologies that may be utilized to transfer and persist the sensor data. To evaluate whether the applications may be used for data collection related to OSA detection, we perform overnight sessions lasting over 8 hours. Data collected from 5 sensors at an accumulated frequency of 46 and 86 Hz resulted in 27.3 and 39.7% of battery usage, respectively. During the longer sessions, we did not experience any disconnections. Therefore, a separate experiment is performed to assess the fault tolerance.

Furthermore, we perform an experiment to evaluate whether the data collected may be exported for further analysis and assessment of sensor accuracy. Our results show that Nyx is able to export data regardless of the frequency and duration of the sessions. Finally, we assess the extensibility of our applications with respect to new sensors. The result of this experiment shows that our smartwatch application may be easily extended with novel sensors without requiring implementation-specific knowledge. Based on the results from our experiments, we conclude that our applications may be used for acquiring sensor data from Fitbit smartwatches for OSA detection.

This page is intentionally left blank.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Thomas Peter Plagemann, and express my sincere gratitude. Your insights and advice at every stage of the research project have been indispensable. Without your guidance, this thesis would not have been possible.

I would also like to thank my family who has supported me during this thesis, but also long before that. Although you do not always understand what I am doing, you are always there to give moral support. For that, I am very grateful.

This page is intentionally left blank.

Contents

List of Figures

List of Tables

Listings

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Motivation

Over the past few years, the market for wearables such as smartwatches and smart bands has grown rapidly and substantially. Despite being a relativity new market, the number of wearable gadgets is expected to reach 500 million units globally in 2023 [2]. Consumer electronics companies such as Apple, Fitbit, and Garmin are competing to deliver devices with novel sensors and features for tracking health and fitness. Similar to the smartphone revolution of the previous decade, smartwatches may revolutionize health monitoring by making it accessible and a common possession for all. Smartwatches are able to monitor a broad spectrum of health metrics using built-in sensors and may thus be used for continuous health monitoring. Recent smartwatches are able to detect heart conditions, e.g., arrhythmia and atrial fibrillation [3], and suggest that proper medical diagnosis is performed by healthcare professionals.

Certain smartwatches are able to measure the oxygen saturation in the blood by utilizing a pulse oximeter sensor. A pulse oximeter uses variations in light absorption to estimate the percentage of oxygen in the blood. Smartwatches with this sensor may be used in monitoring the blood oxygen levels of those who may suffer from chronic sleep disorders like obstructive sleep apnea (OSA). OSA blocks breathing and results in reduced oxygen saturation in the blood. Most cases of OSA are not diagnosed because the patients are unaware of their apnea events [4]. To diagnose a patient with OSA, polysomnography may be performed at a hospital or sleep center [5], which requires that patients are connected to multiple sensors and. In some cases, polygraphy may be performed at home using less equipment [6], but the procedure may still affect the quality of sleep due to the wires. As the adoption rate of smartwatches with pulse oximeter sensors increases, it is likely that considerably more cases will be detected and thereby receive earlier diagnosis and treatment. Thus, devices with such sensors must be tested thoroughly by clinical professionals before being available to the general public and must perform reasonably accurate readings with minimal error.

The availability of low-cost sensors and smartwatches has introduced new directions in the research community. Researchers conduct studies to examine

the accuracy of the sensors and discover novel applications. With that in mind, we consider it an important task to conduct an investigation of the available smartwatches and their ability to record physiological data for further analysis. At the University of Oslo, the CESAR project focuses on lowering the threshold for getting diagnosed with OSA and the proper treatment needed. The project investigates the available sensors and smartwatches in the consumer electronics market that may be used to preclinically detect OSA. The aim of the project is not to diagnose the cases in a clinical fashion, but rather encourage people who may suffer from breathing disruptions to seek medical assistance. The project has developed applications for extracting sensor data from a variety of consumer devices in addition to advanced sensor boards with configurable components. The applications are used to extract data from devices during experiments that can be compared to medical-grade oximeters.

1.2 Problem Description

In our research, we mainly focus on Fitbit Ionic and investigate the development environment which can be used to create an application for collection of sensor data. The built-in sensors of smartwatches are often used for enabling dedicated features and a comprehensive tool for collecting data does, to the best of our knowledge, not exist. For instance, the data from the heart rate sensor is, in addition to being used in exercise tracking, often processed and utilized for evaluating cardiovascular health [7]. Similarly, most smartwatches use accelerometers mainly for measuring physical activity such as step count and calorie consumption. Ultimately, the raw sensor readings are irrelevant for consumers as they do not directly translate to quantitative and qualitative metrics. We are therefore interested in developing an application for the Fitbit Ionic watch that is able to collect data from a multitude of built-in sensors.

The aim of this thesis is to examine the Fitbit Ionic smartwatch with regard to the available sensors and development environment and develop an extensible application for extracting the sensor data for further processing. An application for sensor data acquisition may be useful for recording physiological signals during sleep which, when analyzed, may be indicative of abnormal breathing patterns caused by OSA. The application must be able to record data from sensors throughout a continuous sleep session with minimal disruptions and convey the data to a smartphone. Due to the compact size of smartwatches, the capabilities are limited compared to smartphones and desktop computers. For instance, we must account for less battery life and limited memory and storage. Although the application is designed for the Fitbit Ionic, future generations of Fitbit smartwatches with new or improved sensors should be able to run the application with

few or preferably no modifications. Further, we aim to rigorously test the application and evaluate whether it can be used to reliably collect and transfer the sensor data from the Fitbit Ionic smartwatch to the connected smartphone. Our aims and objectives for this thesis can be summarized as such:

Research Aims

1. Develop an application for the Fitbit Ionic that is able to collect data from the smartwatch while preserving most of the battery for regular use.
2. Develop an application for a mobile platform that is able to
   (a) communicate with the smartwatch application, preferably wirelessly, in order to transfer the sensor data,
   (b) store the sensor data in a structured fashion that can be retrieved and reviewed later, and
   (c) export the data to a computer for post-processing and detailed analysis.
3. Develop extensible applications that can be further augmented with novel sensors and tolerate connectivity loss in case of weak wireless signals.

In order to reach our aims for this thesis, we shall conduct a comprehensive investigation of both platforms, identify the possibilities and immediate challenges. Our investigation should satisfy the following objectives:

Research Objectives

1. Investigate the available tools and application programming interfaces (APIs) that can be utilized for data collection from multiple built-in sensors of the smartwatch in real-time.
2. Explore technologies that can be used for transferring sensor data reliably and wirelessly from the smartwatch to a connected smartphone.
3. Identify suitable data formats which are to be used for data transfer from the smartwatch to the smartphone.

1.3 Contributions

In this thesis, we propose a solution for gathering sensor data from the Fitbit Ionic smartwatch by developing two applications that communicate wirelessly.

The smartwatch application, Hypnos, is responsible for recording the sensor data and transferring them to the connected Android application, Nyx. The Android application is responsible for both storing the data and exporting it for further analysis. The seamless transmission of data between these applications demonstrates that the Ionic smartwatch, as well as other smartwatches from Fitbit, can be used for recording physiological signals.

The applications presented in this thesis are contributions to the CESAR project, which conducts research on developing low-cost solutions for OSA detection. Our findings and results in this thesis are meant to lay the groundwork for further investigation and analysis of the sensor data which may be derived from Fitbit smartwatches.

1.4 Thesis Structure

The thesis is constituent of four main parts, that is, Introduction and Background, Software Development Kits, Design and Implementation, and Evaluation and Conclusion. Presented in detail below is a brief summary of the chapters of each part:

- Part I: Introduction and Background

  Chapter 1, Introduction: In this chapter, we briefly introduce the main topic of the thesis, and the motivation behind the research conducted. We also formulate aims that we intend to satisfy and objectives that will guide the process of our investigation. We then specify the contributions made in this thesis to the CESAR project before summarizing the structure of the thesis, chapter by chapter.

  Chapter 2, Background: In this chapter, we cover relevant background material for our research. First, we briefly explain OSA with regard to diagnosis and oximetry. Then, we summarize relevant master theses which have conducted similar research on smartwatches and sensors related to OSA. Lastly, we summarize the smartwatch alternatives which have been evaluated in this thesis.

- Part II: Software Development Kits

  Chapter 3, Fitbit software development: In this chapter, we present the platform of the Fitbit smartwatch with respect to the operating system and available APIs for acquiring and storing sensor data. We also present available tools used in this thesis for developing an application for the smartwatch.

Chapter 4, Android software Development: Similar to Chapter 3, we inspect in Chapter 4 the mobile platform regarding the tools and concepts employed to design and implement the smartphone application. This chapter summarizes the technologies that enable the communication between the smartwatch and smartphone and familiarize the APIs used in the implementation.

- Part III: Design and Implementation

  Chapter 5, Design: This chapter describes the designs of our applications and the high-level architectural structure that enables reliable data collection. We first define functional and non-functional requirements with respect to the aims and goals of our thesis. Then, we describe the communication details, data transfer techniques, and architectural approaches.

  Chapter 6, Implementation: Chapter 6 comprises the implementation of the smartwatch and smartphone application. First, we present the smartwatch-specific implementations with respect to collecting and handling data from the sensors. Then, we present the relay and permanent storage of this data to the smartphone application.

- Part IV: Conclusion

  Chapter 7, Evaluation: In this chapter, we perform experiments to evaluate our application with respect to the requirements defined in Chapter 5. The experiments should determine whether the proposed solution is able to collect sensor data with minimal and preferably no disruptions.

  Chapter 8, Conclusion: In this chapter, we conclude our work by summarizing the motivation of this thesis and discuss issues that remain unsolved. Finally, we elaborate on aspects that have not been prioritized in our thesis.

Chapter 2

Background

In this chapter, we cover central notions related to our research and introduce related work in the field of physiological computing. We also introduce our wearable alternatives with regard to sleep monitoring.

2.1 Obstructive Sleep Apnea

Obstructive Sleep Apnea (OSA) is a chronic sleep disorder where the upper airways are blocked such that the respiration is interrupted [4]. As a result, the supply of oxygenated hemoglobin in the blood decreases until the person is awake and is able to breathe normally. According to Mayo Clinic, factors such as being overweight or obese, of the male sex, or an older adult greatly increases the chances of being diagnosed with OSA [1]. The World Health Organization (WHO) estimates, in a report released in 2017, that over 100 million are suffering from OSA to some extent on a global basis [8]. Patients that are not diagnosed with OSA may not be aware of their condition and still endure the negative consequences. The consequences are, among other things, sleepiness during the day, snoring during sleep, or lack of concentration [1]. Research has shown that there is a correlation between lack of sleep due to OSA and being involved in traffic accidents [8]. Being able to diagnose more cases would therefore likely mean fewer deaths from accidents due to sleep deprivation.

2.1.1 Diagnosis

Polysomnography (PSG) may reveal whether symptoms experienced are a result of OSA by monitoring the patients while they are asleep. PSG, which is usually performed at hospitals, monitors the heart rate, eye movements, blood oxygens level, movement, and breathing pattern during sleep [5]. Figure 2.1 illustrates an example of the various data gathered during PSG. The diagnosis reveals the frequency of the apnea events as well as the duration. Even though PSG is able to determine cases of OSA without much difficulty, the diagnosis has a long waiting time for potential patients and assumes that the person is well aware that their condition requires a sleep analysis. Furthermore, it requires that patients sleep

in an unfamiliar environment with several wires for the sensors, which in turn may affect the quality of the data being collected.

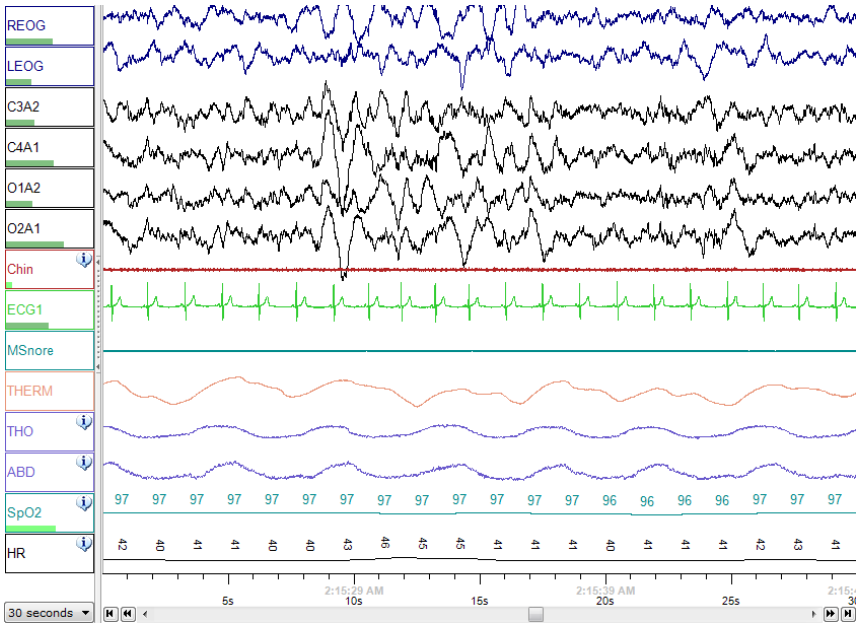


Figure 2.1: Sensor data collected through PSG [1]

Studies with low-cost respiration belts have shown promising results in controlled laboratory experiments [9] and shown in a clinical study that the collected data allows machine learning to properly detect all sleep apnea events [10]. Furthermore, the results from these studies suggest that additional oximeter data might help to improve the sleep apnea severity estimation [11].

2.1.2 Blood Oxygen Saturation

Blood oxygen saturation is a measure of the amount of hemoglobin in the arteries that are bound to oxygen molecules. The typical value of oxygen saturation for a healthy being is approximately 97% [12] but may, however, be within a range of 95% to 100% [13]. A value below would indicate that the body is not able to properly supply oxygen. The oxygen saturation can be measured using a CO-oximeter, which uses a sample of the blood, or a pulse oximeter [12].

During apnea events, the blood oxygen saturation levels drop below what is

[1] By NascarEd - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24506738

considered normal due to suspended breathing. By using an oximeter one can estimate the levels of oxygen in the blood and detect the desaturations that occur during sleep because they may correspond to possible apneas. The precondition is that accuracy of the oximeter is within reason and the desaturations that occur are recorded. Medically-graded oximeters measure the oxygen saturation accurately but are expensive and may be inconvenient to use without medical assistance.

2.1.3 Pulse Oximetry

Pulse oximetry is a non-invasive technique for measuring the oxygen saturation in the blood. The method is based on shining red and infrared light through the skin and measuring the ratio between the wavelengths sent back by oxygen-rich and poorly oxygenated hemoglobins [14]. Since the measurement does not need to take blood samples from the patient, it is a convenient method for monitoring the oxygen saturation over a longer time period.

2.2 CESAR Project

The CESAR project is a research project at the University of Oslo that focuses on lowering the threshold for getting diagnosed with OSA and the proper treatment needed. The project examines low-cost pulse oximeter sensors, smartwatches, smartphones, and other consumer electronics that can be used to detect OSA. The aim of the CESAR project is not to replace the conventional methods used for diagnosing OSA, but rather facilitate the detection of undiagnosed cases with the use of affordable and available technology.

2.3 Related Works

In this section, we present related works which investigate smartwatches and sensors which may potentially be used for OSA detection. The works covered in this section are master theses conducted at the University of Oslo as a part of the CESAR project.

2.3.1 Thesis by H. Bakker

In the thesis ”Sopor: An extensible watchOS application for sleep session recording” [15], Bakker investigates the Apple Watch and whether this smartwatch may be used to collect data during sleep. Bakker considers multiple smartwatch alternatives, e.g., Garmin Fenix 5, Fitbit Ionic, and Apple Watch. The smartwatches

are compared with respect to sensor accuracy, retail price, and development environment. Based on the comparison, Bakker concludes that the Apple Watch delivers the most accurate reading based on previous studies. Therefore, the Apple Watch is investigated further.

Research Aims

The aim of the thesis is to develop an application for the Apple Watch, Sopor, which may be used for collecting data from the various built-in sensors. The application should be extensible with respect to new sensors that may be available in newer generations of the smartwatch. Since smartwatches have limited battery life, the application should use as little battery as possible such that the standard features of the smartwatch are retained. In order to view and analyze the data collected by the Apple Watch application, an additional application should be developed.

Experiments

Various experiments are performed to ensure that the applications fulfill the aims of the research. First, a battery usage experiment is performed to investigate whether the battery usage of the application is acceptable. The requirement is that the battery should deflate no more than 50% during an overnight recording session. Another experiment is performed to determine the CPU and memory usage of Sopor and to ensure that there are no memory leaks in the application. Finally, Bakker performs an experiment that evaluates the extensibility of Sopor. For this experiment, Bakker demonstrates how to add a new `Sink` module to the implementation. A `Sink` is essentially an event handler that is able to perform actions based on the readings received from a sensor.

Main Findings

For certain configurations, e.g., lower sampling rates, the application uses less than 50% of the battery life throughout an 8-hour recording. Also, the application has an insignificant impact on the CPU and memory usage of the smartwatch. Based on results from the experiments performed, Bakker concludes that Sopor is able to collect data from the sensors of the Apple Watch throughout an overnight sleep session. The application may also be extended with additional components to handle the sensor data.

2.3.2 Thesis by K. Frisvold

In the thesis "Non-Invasive Benchmarking of Pulse Oximeters - An Empirical Approach" [16], Frisvold evaluates the accuracy of three inexpensive pulse oximeters in comparison with a medical-grade oximeter by conducting lab experiments.

Research Aims

The main goal of the thesis is to propose a methodology for determining whether or not inexpensive and non-medically-graded pulse oximeters, such as the ones examined by Frisvold, are able to perform on the same level as the NOX T3. The sensors of interest are low-priced and affordable for the general public, but not medically certified to be used in medical diagnosis. By benchmarking these pulse oximeters against a professional and trusted sleep monitor, such as the NOX T3, Frisvold is able to determine their quality with regard to measuring changes in oxygen saturation levels. In addition, Frisvold evaluates whether the observed saturation readings are suited to detect sleep apneas with a high degree of accuracy and precision. The oximeters benchmarked in the thesis are BITalino, an inexpensive and modular board for attaching a range of different sensors, and Cooking Hacks, a more expensive alternative to the BITalino board. Both of the sensors are compared to the medically-graded NOX T3 Portable Sleep Monitor, which is considered the gold standard in monitoring and diagnosing sleep disorders in a medical setting.

Experiments

In order to benchmark the oximeters, Frisvold develops a benchmarking protocol for non-invasive pulse oximeters. The experiments are conducted with 10 subjects that vary in age, sex, and skin pigmentation. To determine whether the oximeters are able to register the saturation and desaturation events of oxygen in the blood, apneas are simulated by breath-holding. Frisvold argues that holding breath will yield variations in saturation similar to the ones caused by *real* apneas. Frisvold also mentions alternative ways of forcing the oxygen saturation to drop by, for instance, breathing in a mixture of gases or breathing inside an enclosed case, but argues that the testing procedure should not cause any health risks to the test subjects. Frisvold concludes that breath-holding is the safest method to achieve a drop in saturation. The subjects were therefore encouraged to hold their breaths but were free to breathe if they felt discomfort.

Main Findings

The results presented in [16] indicate that BITalino is not able to perform adequately as a sleep monitoring device due to the noise in the recorded data. The noise makes it difficult to determine the beginning and end of a desaturation, meaning that data from the BITalino could not be used to detect apnea events. On the other hand, Cooking Hacks was able to determine 63 out of 71 desaturations that were detected by the NOX T3 after recording and analyzing the breathing patterns of the 10 test subjects. Frisvold argues that even if Cooking Hacks was unable to detect 11.3% of the apnea events, the sensor may still be suggestive of some disruptions during sleep.

It is important to mention that eliminating only one of the test subjects significantly impacts the results. By leaving out the last subject, only 3 of the desaturations detected by the NOX T3 are missed, meaning that 95% of the apnea events were detected by Cooking Hacks. Frisvold suggests that this is caused by the last subject which "did not breathe in a steady pattern, and therefore did not have a steady oxygen supply for the hemoglobins to load." Frisvold also emphasizes that the desaturations should reach as low as 70% when the subjects are holding their breaths in order to correctly determine the accuracy of a pulse oximeter. Because most of the subjects were not trained in holding their breath they failed to lower the oxygen saturation. As a result, Frisvold concludes that the benchmarking is only able to give an assurance of the labeled accuracy.

According to Frisvold, the accuracy of the pulse oximeter of Cooking Hacks is labeled 2% and 3% between 100-80% and 80-70%, respectively. The accuracy obtained from the experiments is 1.34% for all the subjects, in other words well within the reported value. Due to the noisy data of BITalino, no comparison has been done with respect to the simulated apnea events, thus no accuracy measure for BITalino is provided.

2.3.3 Thesis by F. Halvorsen

In the thesis "Garmin smartwatches to detect desaturation events as part of OSA screening at home" [17], Halvorsen investigates the use of smartwatches by Garmin for detecting OSA by comparing them against a gold standard sleep monitor. The research is a part of a master thesis in the context of the CESAR project.

Research Aims

The aim of the research is to figure out to which extent the oximeter-enabled smartwatches by Garmin are able to detect apneas and variations in oxygen saturation as well as determining their accuracies. Halvorsen compares four Garmin smartwatches with the NOX T3, which serves as a gold standard for the investigation. For acquiring data from the pulse oximeter sensors in the smartwatches, Halvorsen develops an Android application using the Garmin SDK. The smartwatches examined are the Venu, Vivosmart 4, Vivoactive 4, and Fenix 6 Pro. Although the smartwatches vary in price they all come with a built-in pulse oximeter sensor. The oximeter sensors are benchmarked by conducting lab experiments.

2.3.4 Lab Experiments

As a part of benchmarking the smartwatches, Halvorsen conducts four types of experiments: a user acceptance experiment of the Android application for acquiring data, a connection loss experiment, an energy usage experiment, and an oximeter sensor experiment where the quality of the oximeters are evaluated. In addition, a longer experiment of the oximeters is performed in a less controlled environment with only one individual. Below, we briefly summarize each test:

- **User Acceptance Experiment:** This experiment evaluates the usability of the Android application by conducting acceptance tests with respect to efficiency, effectiveness, errors, and user satisfaction. The subjects in the test population have varying degrees of proficiency regarding the use of mobile devices.
- **Connection Loss Experiment:** In this experiment, Halvorsen tests the ability of the smartwatches and the Android application to handle disconnections that may occur. Because mobile devices are prone to disconnections, it is important to test whether the devices are able to handle connection loss.
- **Energy Usage Experiment:** Mobile devices have limited energy and keeping an active communication between a smart watch and an Android application for hours could drastically impact energy usage. An application that drains the battery of the phone would limit the user from monitoring the oxygen saturation throughout the night. Therefore, Halvorsen measures the energy impact of the application when it is communicating with the smartwatches.

- **Short Oximeter Experiment:** The short experiment is carried out with a population of 8 subjects. To acquire deep desaturations the subjects are, in contrast to Frisvold's experiments, trained in breath-holding before the actual experiments take place. In this experiment, all of the subjects wear the Venu and Fenix 6 Pro for a short period of time, during which they simulate apneas.
- **Long Oximeter Experiment** In this experiment, the Venu and Fenix 6 Pro are tested during a sleep session. The test lasts approximately 6 hours and is conducted by only one individual. The purpose of the test is to examine the performance of the oximeters in a setting where external factors may affect the testing procedure. Because this test is conducted during sleep, the desaturations are not produced by breath-holding but expected to occur arbitrarily.

Apnea Detection Procedure

Halvorsen defines an apnea event as a desaturation equal to or greater than 3%. The desaturation events from the Garmin smartwatches are compared against the desaturations of the NOX T3. Using the built-in accelerometer from both devices, the oximeter data is synchronized such that the desaturations from the NOX T3 are aligned with the desaturation of the smartwatches. A sudden movement in the beginning or at the end of a recording ensures that both devices have a notable spike in the accelerometer data which can be used to correctly align the oximeter readings. This technique does not, however, consider the fact that the sampling rate of the sensors may differ. Therefore, Halvorsen performs additional resampling of the data. Furthermore, a Python script is used for obtaining the total count of desaturation events. The script loops through the data stream and counts the total amount of desaturations equal to or beyond 3%. A variation in oxygen saturation is regarded as an apnea event only if it is lower than the mean value of the last 120 seconds.

2.3.5 Main Findings

Halvorsen develops a wrapper application for the Android platform which is able to communicate with all of the Garmin smartwatches via Bluetooth. The application is able to persist the oximeter data to a file that can be exported and then used for the benchmarking procedure. All subjects that tested the application managed to complete the tasks without much effort. Based on the user acceptance test, Halvorsen concludes that the application is usable both for novice and advanced users.

Initially, the objective was to benchmark the oximeter of all four Garmin smartwatches. After conducting the preliminary experiments, the Vivosmart 4 watch and the Vivoactive 4 watches were excluded from the oximeter testing procedure. The reason behind this is that these watches yielded inconsistent sensor readings. Since both the Venu and Fenix 6 Pro performed reasonably well, further benchmarking and comparison with the NOX T3 were only carried out using these.

Oximeter Accuracies

In the shorter experiment conducted with the 8 test subjects, the NOX T3 detected 24 desaturation events of which the Fenix 6 Pro only detected 5. In other words, the Fenix 6 Pro is only able to detect approximately 20% percent of the total desaturation events. On the other hand, Venu observed 14 desaturations of the 24 detected by the NOX T3 yielding an accuracy of around 58%. Halvorsen suggests that the results may be due to the oximeter of the smartwatches having a lower sampling rate compared to the NOX T3 and that the script used to detect the desaturations may be counting fewer than the watches detect. In the long-running experiment, the NOX T3 detected 3 desaturation events. This value is significantly lower compared to the previous test because the test subject did not force the oxygen saturation to drop by breath-holding. Instead, the desaturations occurred unconsciously during sleep. The smartwatches were able to identify all of the desaturations detected by the NOX T3 but they also detected desaturations that were false positives. Specifically, the Fenix 6 Pro detected 2 false positives and the Venu detected 6. Halvorsen concludes that the smartwatches are inconsistent with regard to detecting desaturations and that the accuracy varies greatly between experiments. Also, Halvorsen suggests that the Fenix Pro 6 might be better suited for detecting more frequent desaturations as opposed to the Venu watch.

2.4 Summary of Related Works

The research conducted in [16] uses software developed in a previous thesis to gather data from the oximeter sensors and assess their accuracy. Furthermore, [17] develops software to acquire oximeter sensor data and evaluates the quality of the sensors. Although we wish to conduct similar research for the Fitbit Ionic regarding its ability to detect apnea events and variations in oxygen saturation, since the oximeter is not open for third-party use, we are not able to evaluate the accuracy of the sensor. Instead, we investigate the Fitbit platform and design and implement a data acquisition application that may be used to collect data

from the built-in sensors. When the oximeter sensor is enabled, the application may be used to collect data from and assess the quality of the sensor. Therefore, we shall conduct research similar to [15] by laying the groundwork for sensor data collection and perform experiments to evaluate the solution by using the other sensors.

2.5 Smartwatch Alternatives

In this section, we present the smartwatch alternatives which have been evaluated in our thesis. Since some alternatives have already been evaluated in the CESAR project, we only focus on the Fitbit Ionic and the Withings ScanWatch. In Sections 2.5.1 and 2.5.2, we cover the key features and qualities of the Fitbit Ionic smartwatch and the Withings ScanWatch, respectively.

2.5.1 Fitbit Ionic

Fitbit Ionic is a smartwatch developed by Fitbit Inc. released in October of 2017 and marketed as a health and fitness tracker [18]. The watch houses a wide variety of built-in sensors, e.g., accelerometer, barometer, gyroscope, optical heart rate sensor, ambient light sensor, and temperature sensor [19]. By making use of the aforementioned sensors, it is able to track step count, calorie burn, sleep, and continuous heart rate and propagate this information to the smartphone that it is connected to. Fitbit Ionic is mainly targeted towards consumers who are interested in tracking their fitness activities. It also provides additional smartwatch features, e.g., an application store for downloading third-party applications, media playback over Bluetooth, and contactless payment via NFC. Figure 2.2 shows the small form factor of the Fitbit Ionic smart watch.

The watch also includes a built-in pulse oximeter that is able to measure the oxygen level in the blood. Although the oximeter sensor was shipped with the device, it is not activated by Fitbit until as recently as January 2020 [20]. The reason for the delayed activation of the oximeter sensor is unknown but Fitbit is still awaiting clearance by the FDA in order to use its fitness trackers for sleep apnea detection [21].

2.5.2 Withings ScanWatch

ScanWatch is a smartwatch developed by Withings and launched in September of 2020. Withings is promoting the ScanWatch as a *hybrid smartwatch* [22] since

[2] `https://www.amazon.com/Fitbit-Ionic-Smart-Charcoal-Included/dp/B074VDF16R`

Figure 2.2: Fitbit Ionic [2]

the watch face resembles an ordinary watch and the smart features are accessed via a 0.54-inch circular LED screen on the top half of the watch. Unlike the Ionic, the screen does not take touch input and the menus are navigated using a rotary crown on the side of the watch which acts as the main input for the smartwatch. As can be seen in Figure 2.3, the Scanwatch is similar to a traditional watch.

Similar to the Fitbit Ionic, the ScanWatch has a built-in accelerometer and altimeter for tracking physical activity and exercises [22]. Additionally, the ScanWatch houses a multi-wavelength photoplethysmography (PPG) sensor for measuring heart rate, oxygen saturation as well as detecting irregular heart rhythms. Withings ScanWatch has been clinically validated and obtained a CE clearance for use in Europe. During our research period, the ScanWatch has yet to obtain clearance from the U.S. Food and Drug Administration (FDA). However, Withings has expressed that the process is underway [23].

2.5.3 Comparison of Smartwatches

In our research, we have evaluated the Fitbit Ionic and the Withings ScanWatch. These smartwatches have numerous built-in sensors which are used to track fitness activities and overall health. Most of the sensors are available in both of the smartwatches, e.g., accelerometer sensor, altimeter sensor, optical heart rate sensor, and pulse oximeter sensor. The Fitbit Ionic houses two additional sensors; a gyroscope and a temperature sensor [24]. Our investigation shows that the

[3] `https://www.withings.com/de/en/scanwatch`

Figure 2.3: Withings ScanWatch [3]

temperature sensor is not available through the SDK and may therefore not be used in this thesis. In Table 2.1, we list the specifications of both smartwatches.

Pulse Oximeter Data

Despite the fact that both the smartwatch alternatives include a pulse oximeter sensor, there exists, to the best of our knowledge, no research and investigation available with regard to their accuracies. The Fitbit Ionic was introduced in late 2017 but the sensor was not activated at launch. In early 2020, Fitbit announced that the Ionic will be able to monitor the variation oxygen saturation. However, monitoring the oxygen saturation is only available when the watch detects that the user is sleeping and the raw data is not directly available per se. In the Fitbit mobile application (FMA), a graph showing the variations in oxygen saturation is available for each sleep session. The graph, of which an example can be seen in Figure 2.4, shows the changes in oxygen saturation recorded by the smartwatch but without any indication of the actual values. Further investigation has revealed that the data points used to create the graphs are available by exporting data from the associated Fitbit user account. Surprisingly, the exported data does not contain the oxygen saturation as percentage values but rather values measured by the sensor of the smartwatch. Therefore, we are not able to assess the quality of the oximeter sensor in this thesis.

The ScanWatch is a relatively new smartwatch and has therefore not been tested thoroughly by the research community. The oximeter sensor in the Scan-

Table 2.1: Comparison of the Fitbit Ionic and Withings ScanWatch

	Fitbit Ionic	**Withings ScanWatch**
Launch Date	October 2017	September 2020
Accelerometer	Yes	Yes
Altimeter	Yes	Yes
Gyroscope	Yes	–
Optical Heart Rate Sensor	Yes	Yes
Pulse Oximeter	Yes	Yes
Temperature Sensor	Yes	–
ECG	–	Yes
GPS	Yes	–
Bluetooth	Yes	Yes
NFC	Yes	–
Screen	1.5 inch (348 x 250)	0.54-inch (116 x 80)
Weight	50g	58g
Operating System	Fitbit OS	Undisclosed
SDK Support	Yes	–
API Support	Yes	Yes
Battery Life (Self-reported)	5 days	30 days
Price at Launch	$300 or €350	€279

Watch is enabled and is available for on-demand readings as well as continuous readings throughout a sleep session. Users are able to take a measurement on-demand by keeping still for 30 seconds. Continuous readings are performed automatically when the sleep analysis feature is enabled. Similar to Fitbit Ionic, the readings are not available directly to the user. In the Withings Health Mate mobile application, the breathing disturbances are charted, which uses the oximeter sensor to detect changes in oxygen saturation.

SDK Support

Since the ScanWatch is a hybrid between a regular watch and a smartwatch, it is missing functionality that is typically present in a smartwatch. For example, it is not possible to install third-party applications via an application store. Instead, the ScanWatch comes with preinstalled applications for tracking physical activity, measuring oxygen saturation, and performing electrocardiography. Currently, Withings provides no SDK for application development for third-party developers but this may change in the future via a software update. In order to retrieve data from the smartwatch, Withings provides a RESTful API for developers. Unlike

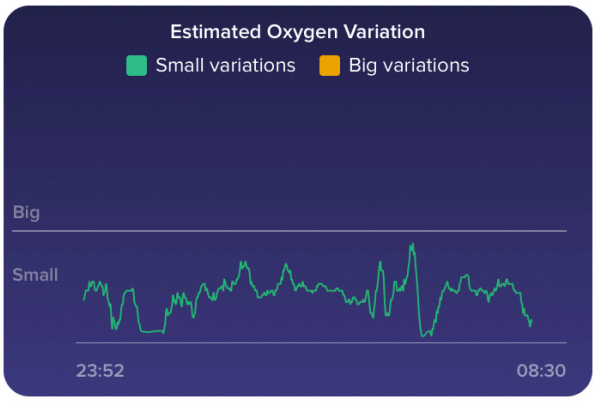


Figure 2.4: Estimated Oxygen Variation graph recorded by the Fitbit Ionic.

the Fitbit Ionic, there is no API available for full sensor control. Since we are not able to create a custom application to acquire sensor data from ScanWatch, using this smartwatch is not feasible in our research. On the other hand, the Ionic smartwatch allows third-party applications and provides APIs to access the sensors. Therefore, we examine the development environment of the Fitbit Ionic to determine whether it may potentially be used to collect data that may be useful for OSA detection.

Part II

Software Development Kits

Chapter 3

Fitbit Software Development

In this chapter, we present the Fitbit Software Development Kit (SDK) by exploring the development environment and platform tools for Fitbit smartwatches. We also present the available APIs that enable the acquisition of sensor data from the Fitbit smartwatch.

3.1 Operating System of Fitbit Ionic

Smartwatches and fitness trackers by Fitbit are running Fitbit OS, which is a closed-source and proprietary operating system developed by Fitbit. The first version of the OS was launched alongside the Fitbit Ionic smartwatch in 2017 [18] and has later been made available for devices launched subsequently. The OS is updated regularly and the Ionic used in our research is running version 27.72.1.9, the latest version available. Smartwatches and fitness trackers from Fitbit prior to 2017 do not support Fitbit OS and the tools for developing third-party applications. The Fitbit Ionic is therefore the first smartwatch from Fitbit that enables researchers and software developers to utilize the built-in sensors by building custom applications.

The user interface (UI) of Fitbit OS is mainly based on touch input as the Ionic smartwatch has a capacitive touch screen and few physical buttons. The one button on the left is used to navigate backward in the UI and the two buttons on the right can be assigned to shortcuts for frequently used features or applications. The clock face of the watch, the screen that is shown when the device is raised up, can be customized by downloading third-party faces from the application store. Swiping down from the top, brings the recent notifications, and swiping up from the bottom brings up the activity metrics, eg., steps, calories burnt, and distance. Preinstalled and third-party applications are available by swiping left and are organized in an endless grid. In Figure 3.1, we illustrate the general structure of the OS. An arrow represents a swipe gesture for navigating to that screen. As can be seen in this figure, third-party applications are available a swipe from the main clock face.

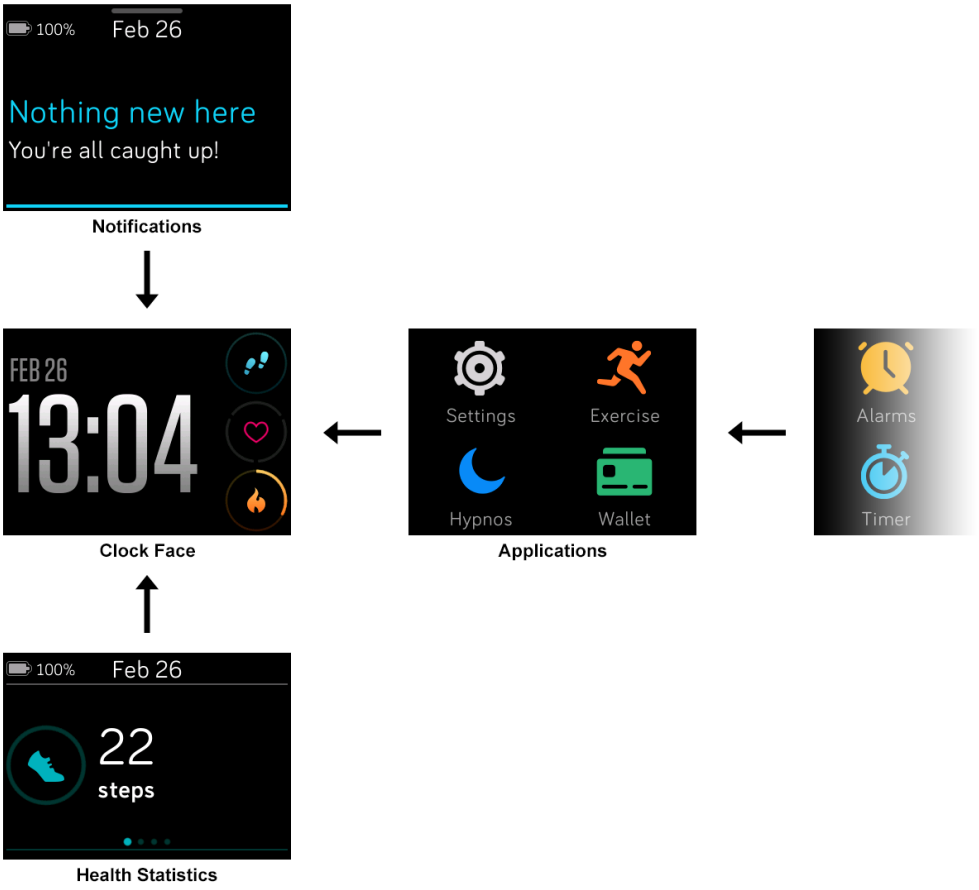


Figure 3.1: UI navigation on Fitbit OS

3.2 Fitbit OS SDK

Developing applications for Fitbit devices is similar to mobile development. Applications can be created by utilizing the integrated development environment and tested on a simulator or a physical device. The SDK is free and developers may publish applications to an application store for distribution. Parts of the SDK are open-sourced and publicly available for developers to suggest changes or issue bug reports [25]. The SDK delivers tools for developers for creating clock faces and applications for a wide range of Fitbit devices, including the Fitbit Ionic watch, and is constituent of three main components:

- **Device API** provides access to the hardware components of the Fitbit OS devices,

- **Companion API** for accessing the resources of the connected mobile phone, and
- **Web API** for accessing fitness and health-related data gathered by Fitbit devices and stored on remote servers.

The Device API gives access to the hardware components of the Ionic smartwatch by abstracting them through objects and interfaces. For instance, to develop our data acquisition application we must utilize the Device API to access various hardware sensors that can be used for recording sleep sessions. The Device API is also necessary for storing and retrieving files and controlling the hardware buttons of the smartwatch. In Figure 3.3, the comprehensive architecture of the Fitbit SDK is illustrated with the main components, the underlying components, and the intercommunication between. As can be seen from this figure, certain APIs are available on both components to enable communication and data transfer.

The Companion API provides access to the available resources of the connected smartphone through the FMA. The smartwatch is able to, for instance, use the cellular connectivity of the phone to fetch data from the Internet, despite the fact that it does not have built-in cellular reception. Unlike the Ionic smartwatch which features a built-in GPS, there are some models that do not have an integrated GPS that can, with the use of the Companion API, utilize the GPS on the smartphone. On the one hand, the Companion API enables the smartwatch to leverage additional functionality from the smartphone. On the other hand, the application developed for the smartwatch becomes dependent on the smartphone being available. When installing an application on the Ionic smartwatch, modules that use the Companion API are installed within the FMA on the smartphone. The FMA, which can be seen in Figure 3.2, is developed by Fitbit and is used for accessing fitness activity data and other metrics from Fitbit devices like the Ionic smartwatch. In other words, because parts of the smartwatch application are running inside the FMA, it is required that the FMA is running in the background for any Fitbit application that uses the Companion API.

The Web API provides access to data from all Fitbit devices, from smartwatches to smart scales. The data can be accessed either using Fetch API in the Companion API (see Figure 3.3) or through a separate REST API. Applications that use the Web API must be registered with Fitbit since it enables third parties to access all health-related data of the users. The Web API provides access to

[4] `https://play.google.com/store/apps/details?id=com.fitbit.FitbitMobile&hl=no&gl=US`

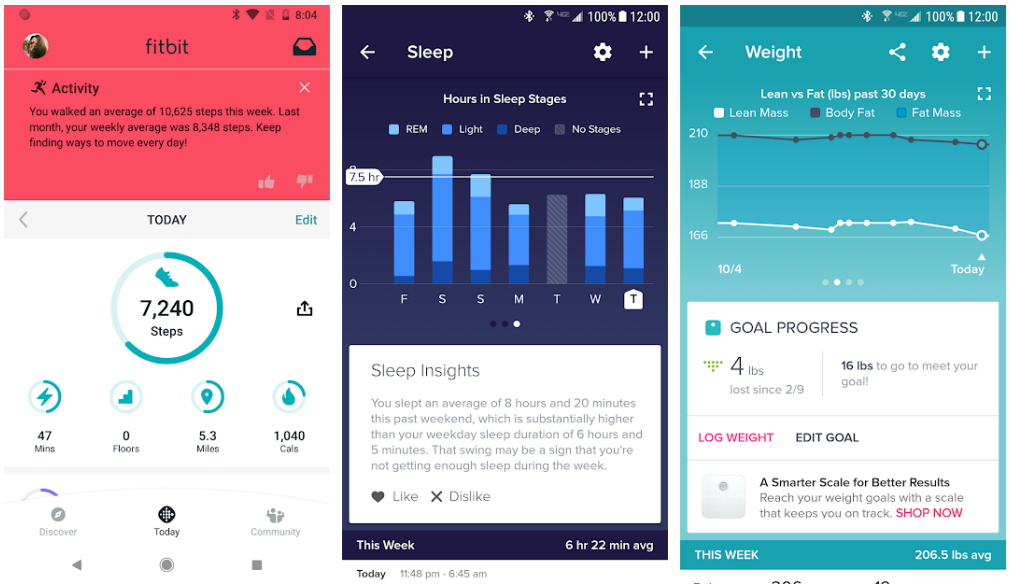


Figure 3.2: FMA running on Android [4]

heart rate and physical activity metrics, e.g., step count, distance, and elevation, by using appropriate endpoints listed in the developer references. However, it is important to note that the data is mostly processed, i.e., raw data from the sensors are not available. The only raw sensor data that is accessible with the Web API is heart rate, but even this data is limited to a minute interval. API requests using the Web API is limited to 150 requests per hour [26], and so even if other sensors were available, we would need more requests to gather data from all of them. For this reason, the Web API is neither suitable nor practical to be utilized in our data acquisition application.

3.2.1 Fitbit Studio

Fitbit Studio is a web-based development environment (Web IDE) for developing applications for a range of Fitbit smartwatches. There are both advantages and disadvantages of using a Web IDE. A clear benefit is the ability to run the development environment on any device or configuration that has support for web browsing (assuming that JavaScript is supported). A disadvantages of a Web IDE is that it requires stable network connectivity to transfer applications over-the-air to the smartwatch. Furthermore, since the development occurs inside a web browser, it is not possible to handle separate versions using a version control system (VCS).

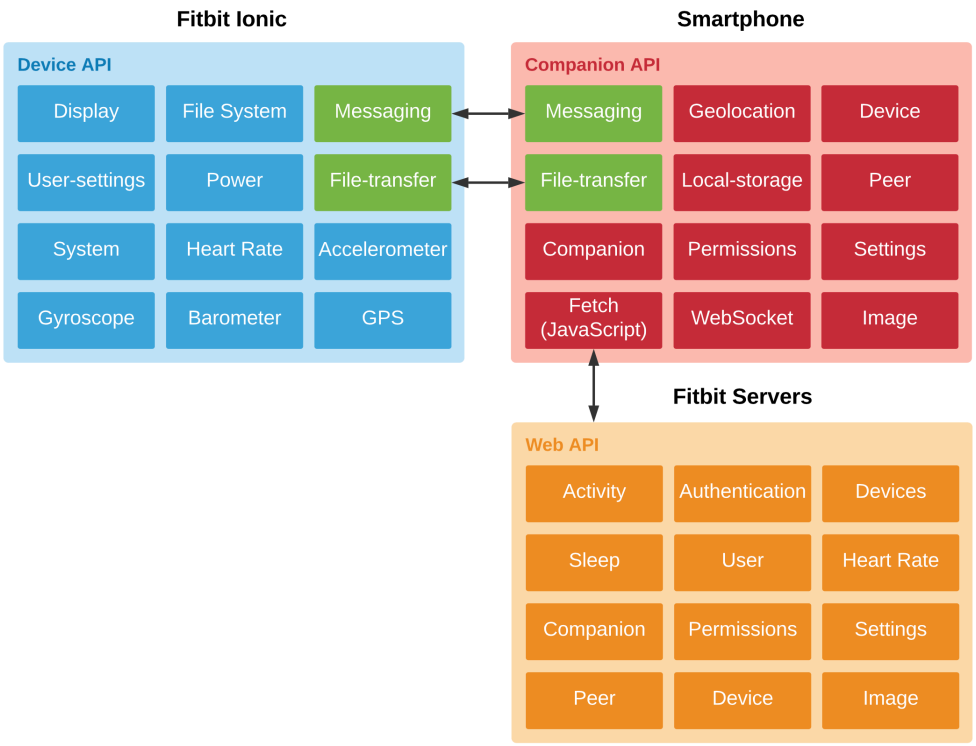


Figure 3.3: Macro architecture of components of Fitbit OS SDK

3.2.2 Command Line Interface

An alternative way of developing applications for the Fitbit Ionic is using the Command Line Interface (CLI) for the Fitbit SDK. The CLI uses Node.js, a JavaScript runtime environment [27], to create scaffold projects for developing new applications. As opposed to the Web IDE, using CLI provides the ability to use the preferred editor of choice. Also, since the development is local, we may use a VCS to keep track of separate versions. Applications developed must be installed on the device using the Developer Bridge. The Developer Bridge uses the onboard WiFi of the Ionic smartwatch to transfer and install the application [28]. Although the CLI allows offline development, installing applications for testing requires an active Internet connection.

3.3 Fitbit Development

Development for the Fitbit platform is similar to conventional web development but accommodated for a smaller device with limited capabilities. JavaScript is the

Table 3.1: Project directory structure for a Fitbit application

Directory	**Contents**
/app	Source code running on the smartwatch and utilizing the Device API.
/companion	Source code running within the FMA and utilizing the Companion API.
/common	Source code and resources available from both /app and /companion.
/resources	Resource files, e.g., CSS, GUI and image files.
/settings	Directory for files that use the Settings API.

main development language and is used for programming both the smartwatch application and companion. The source code is separated into multiple directories corresponding to the components of the SDK. Application code running on the Ionic smartwatch must be placed in the `/app` directory. A Javascript file named precisely `index.js`, must be created for the application to execute. This file acts as an entry point in which we may reference other classes and files within the `/app` directory. Source code that runs within the FMA and utilizes the Companion API must be located within the `/companion` directory. Similar to the application code, an entry file must be present for the companion application to start. Further, a directory named `/common` should contain source code used by both the smartwatch and companion application in order to reduce code duplication.

Views are defined as `.gui` files using SVG, a markup language that resembles XML. These files consist of components in a tree structure that build up the UI of a Fitbit application. Similar to web development, styling is defined in CSS and can be used within SVG components as well as JavaScript code. Resource files, such as CSS, GUI files, and images, must be placed under the `/resources` directory and a file `index.gui` must be placed within this folder as an entry point. Inside the `index.gui`, we may reference other GUI files. In Table 3.1, we summarize the directories of a typical project and the purpose of each directory.

3.3.1 Device API

The Device API provides access to the built-in sensors, file system, and other hardware components of a Fitbit smartwatch. In this section, we present the APIs that are useful to our application that should be able to collect data from the hardware sensors. Although the Device API provides APIs for each sensor, we present only the Accelerometer API since they all function similarly.

Accelerometer API

The Accelerometer API allows developers to access readings from the built-in accelerometer sensor. This sensor measures the acceleration in the x, y, and z-axes. As can be seen from Figure 3.4, the z-axis is aligned orthogonally with the screen and is pointing towards the user, whereas the x and y axes point to the right and up, respectively. The measurements include the gravitational acceleration of 9,81 m/s^2. In other words, the acceleration in the z-axis will be equal to the gravitational acceleration when the smartwatch is lying flat on a table.

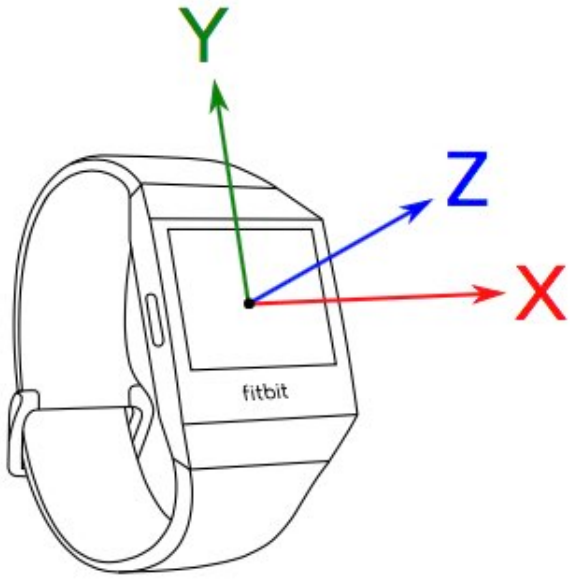


Figure 3.4: Fitbit Ionic measures acceleration in 3 axes, x, y, and z [5]

The accelerometer sensor is able to deliver readings up to 100 times a second. However, the supported frequencies vary between the sensors, e.g., the heart rate sensor only allows sampling at 1 Hz. Sensor readings can be delivered one by one or in batches containing multiple readings. Continuous monitoring of the sensor data will impact battery life if the readings are processed individually. Furthermore, batched readings will decrease resource usage by only handling the sensor data at certain intervals and thereby increase the battery life and CPU usage. For our application that should continuously collect sensor data, it is crucial that the process does not deflate the battery in the midst of a session.

[5] `https://dev.fitbit.com/build/guides/sensors/accelerometer/`

In theory, batched readings should contribute to reduce the battery usage of the sensors.

In order to receive accelerometer data, we first need to import the `Accelerometer` class and instantiate an `Accelerometer` object. The frequency of the readings and the batch size can be determined by passing a configuration object with integer values for each option. We then assign the callback function by adding an event listener to the `Accelerometer`-object and handle the readings within the callback function. At last, we must call the `start`-method of the `Accelerometer` in order to start the sensor and receive readings. In Listing 3.1, we show how to subscribe to accelerometer readings as described above. The process is similar for the other sensors where the only difference is the name of the properties being accessed. Since the accelerometer sensor measures tri-axial acceleration, we should access the x, y, and z properties of the sensor object. These properties depend on the sensor API being used.

Listing 3.1: Using the Accelerometer API

```
1  import { Accelerometer } from "accelerometer";
2
3  const config = {
4      frequency: 1,
5      batch: 1
6  }
7
8  const accelerometer = new Accelerometer(config);
9
10 accelerometer.addEventListener("reading", () => {
11     console.log(
12         `ts: ${accelerometer.timestamp},
13         x: ${accelerometer.x},
14         y: ${accelerometer.y},
15         z: ${accelerometer.z}`
16     );
17 });
18
19 accelerometer.start();
```

File System API

The File System API (FS API) is a file handling API that enables both reading from and writing to files located on the internal storage of the smartwatch. Additionally, we may also rename, delete, and retrieve the metadata of a file. Each application on the Fitbit Ionic has its dedicated directory for storing application-related files. The applications are *sandboxed*, i.e., they cannot read files or write to files of other applications despite being stored on the same device [29]. All

of the operations performed by the FS API are synchronous, i.e., the read or write operation must be completed before continuing the execution. Therefore, performing long-running operations, e.g., retrieving a large file, will halt the system as the I/O operations are blocking. In our application, the FS API may be utilized to write the sensor data or configuration files to disk.

With the FS API, we are able to save and retrieve files in a variety of formats, e.g., ASCII, JSON, UTF-8, and binary files either encoded as CBOR or as raw bytes. It is worth noting that the Ionic has 2.5 gigabytes of storage available, but third-party applications are limited to 15 megabytes of storage for local files [30]. This limit is imposed for both files written with the FS API as well as the application source code. Also, since the API does not support creating directories, all files read or written must be located within the `/private/date` directory. Below, we list the available methods of the FS API with examples for each method:

- **Listing Files:** To list files in the file system, we can use the `listDirSync` method and pass the default directory, `/private/date`, as a parameter. The method returns an iterator that can be used to cycle through individual files within the directory. In Listing 3.2, we exemplify how to list files.

Listing 3.2: Listing files in a directory using the FS API.

```
1 import { listDirSync } from "fs";
2
3 const listDir = listDirSync("/private/data");
4
5 while((dirIter = listDir.next()) && !dirIter.done) {
6     console.log(dirIter.value);
7 }
```

- **Writing JSON, UTF-8, and ASCII Files:** To write JSON, UTF-8, and ASCII files, we may import and use the `writeFileSync` method and pass three parameters. The first parameter determines the name of the file, the second is the data to be written, and the last parameter is the file format. In Listing 3.3, JSON data is written to disk using the `writeFileSync`-method. If the file already exists, this method will overwrite the file and replace its contents.

Listing 3.3: Writing JSON data to a file using using the FS API.

```
1 import { writeFileSync } from "fs";
2
3 let data = {"name": "John", "age": 24};
4
5 writeFileSync("data.json", data, "json");
```

- **Writing Binary Files:** Writing a binary file is more complicated since we need to allocate the exact number of bytes needed to store the data. First, an `ArrayBuffer` is created by specifying the total bytes. Then, we create a typed array and insert values to the `ArrayBuffer` using the indices of the array. Writing a binary file is demonstrated in Listing 3.4.

Listing 3.4: Writing a binary file using the FS API

```
1  import { openSync, writeSync, closeSync } from "fs";
2
3  let file = openSync("filename.bin", "w+");
4  let buffer = new ArrayBuffer(3);
5
6  let bytes = new Uint8Array(buffer);
7  bytes[0] = 1;
8  bytes[1] = 2;
9  bytes[2] = 3;
10
11 writeSync(file, buffer);
12 closeSync(file);
```

- **Reading JSON, UTF-8, and ASCII Files:** Reading files of the JSON, UTF-8, or ASCII format is similar to the `writeFileSync` as it only requires two parameters: the name of the file and the format. An example of reading a JSON file is demonstrated in Listing 3.5.

Listing 3.5: Reading a JSON file using the FS API.

```
1  import { readFileSync } from "fs";
2
3  let jsonObject = readFileSync("data.json", "json");
4  console.log("Sensor Identifier: " + jsonObject.sensorIdentifier);
```

3.3.2 Companion API

The Companion API allows application developers to take advantage of resources available on the connected smartphone, e.g., by allowing access to the GPS and Internet through the Geolocation API and Fetch API, respectively. The API can for example be used to monitor the location changes of the phone and notify the watch if the location change is significant.

WebSocket API

In the companion application, we may establish WebSocket connections to remote or local WebSocket servers. A WebSocket is essentially a protocol allowing bidirectional messages between two machines. Using this API, the companion may

send data to and receive data from a WebSocket server using the smartphone. The underlying protocol is explained in detail in Section 4.3.

3.3.3 Shared APIs

Some APIs are available for both the smartwatch and the companion. These APIs are typically related to communication as both endpoints must be able to understand and act upon the messages.

Messaging API

The Messaging API allows communication between the smartwatch and the companion. The API has support for sending data from the watch to the phone and vice versa. This enables the smartwatch to gather information from the built-in sensors and to send them to the Fitbit application on the smartphone or to request data from the smartphone.

File-transfer API

The File-transfer API is responsible for file transfers to and from the connected smartphone. As described in the developers guide the File-transfer API "allows developers to easily and reliably transfer binary and text files from the mobile companion to the device, even if the application is not running on the device" [29]. For this purpose there are two queues, an `outbox` queue for sending files and an `inbox` queue for receiving the files from the `outbox` queue.

Chapter 4

Android OS and SDK

In this chapter, we present the Android operating system and software development kit for Android. The aim of this chapter is to give a brief overview of the concepts and APIs used to develop an application for persisting sensor data.

4.1 Android OS

Android is a free and open-source operating system introduced in 2008 and designed primarily for smartphones and tablets with touch screens [31]. The operating system is maintained and developed by Google but is owned by the Open Handset Alliance, a group of several mobile phone manufacturers, mobile operators, semiconductor companies, and software companies [32]. Thus, Android is able to run on a wide variety of devices with varying hardware components. A key benefit of the operating system is the fact that an application written for Android will run on multiple devices from different manufacturers without any additional effort from the software developers. Since a large percentage of the smartphones in use today are running Android [33], developing an application for the Android platform will reach a fairly large portion of the smartphone market. By developing the smartphone application for our data acquisition application for the Android platform, we are able to run the application on almost any smartphone.

4.2 Android SDK

The Android SDK enables developers to create applications for Android. In this section, we briefly present APIs and concepts which are relevant to the implementation of our Android application, such as Activities, Services, and database solutions. Android applications may be developed using Java, Kotlin, and C++. However, it is possible to develop an application by using only Java.

4.2.1 Activity

An `Activity` in an Android application corresponds to the view that the user may interact with, for instance, by tapping a button on the screen. Applications with multiple views are made up of multiple activities where the navigation between them is handled by the OS. In Android, activities are organized in stacks, i.e., the most recently launched `Activity` is located at the top, and a designated back button pops the topmost `Activity` off the stack [34].

In order to create an `Activity`, a Java class must be created that extends the `Activity` class (alternatively `AppCompatActivity` class for backward compatibility with older Android versions) and its methods must be implemented. These methods are called automatically when the `Activity` changes state from one to another due to navigation in the UI.

Listing 4.1: Creating an Activity by attaching a view.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     // Additional initialization
7 }
```

The `Activity` class acts as a controller and is responsible for the behavioral aspects of the view. The UI components for the `Activity` are declared in a separate XML file where the components, such as buttons, text input fields, etc., are organized in a tree structure. When the `Activity` is created, the onCreate method is called. The `onCreate` method is one of the methods that must be implemented and is responsible for initializing the `Activity` before it is visible on the screen. As can be seen in Listing 4.1, the view components in the XML-file, `activity_main`, are attached to an `Activity` by the `setContentView` method.

Activity Lifecycle

Every `Activity` created follows a particular sequence of states that is called the *Activity Lifecycle*. Activities change state throughout their lifespans depending on their visibility on screen. Appropriate methods are called between these changes to handle different states. Below, we summarize the states and the methods that are called prior to entering a new state:

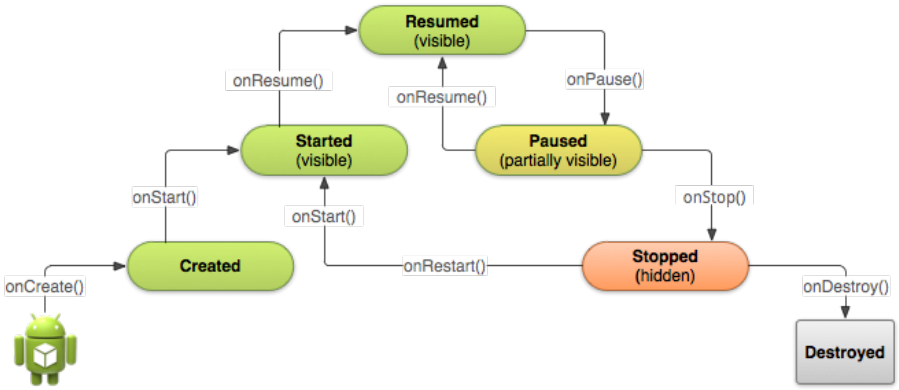


Figure 4.1: The activity lifecycle visualized using a state diagram [6]

- **Created:** Before an `Activity` is visible, the `onCreate` method is called and the initial setup is performed. As can be seen in Figure 4.1, this is the entry point of the `Activity`. This method leads to the first state for the `Activity`, that is, the `Created` state. In this state, the view hierarchy is attached but not visible on the screen and may therefore not respond to user input.

- **Started:** In the `Started` state, the view of the `Activity` is attached and visible on the screen but not able to respond to user input. An `Activity` may be in this state after being created as well as being started again after being in the Stopped state.

- **Resumed.** The `Resumed` state is characterized by an `Activity` that is running in the foreground and able to respond to user interaction. Unlike the `Created` and `Started` states, an `Activity` may be in this state indefinitely, as long as the memory of the system allows it.

- **Paused:** Whenever an `Activity` is partially obstructed by another view but not completely invisible to the user, it is in the `Paused` state. For example, when the user is presented with a dialog where action has to be taken, the `Activity` below is temporarily disabled. From the `Paused` state the `Activity` may go back to the `Resumed` state or the `Stopped` state.

- **Stopped:** If the `Activity` is no longer visible on screen it is in the `Stopped` state. From this state, the `Activity` can return to `Started` and then

[6] `https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-1-get-started/lesson-2-activities-and-intents/2-2-c-activity-lifecycle-and-state/2-2-c-activity-lifecycle-and-state.html`

Resumed state and once again receive user input, or it can be destroyed either because the application is quit or the system needs memory for a process with higher priority.

- **Destroyed:** The Destroyed state clears the Activity from memory and removes it from the view stack. If the user navigates back to the Activity it will be recreated. The onDestroy method may be implemented if any operations should be performed before termination of an Activity. As opposed to desktop operating systems, Android may destroy an Activity if it is using too much memory or a more important task is in the foreground.

4.2.2 Services

A Service is a long-running process designed to perform a specific task in the background without requiring user input. Unlike an Activity, Services are not visible in the UI since they are not possible to interact with. Services have their own lifecycle similar to the Activity lifecycle. In Android, there are two main types of services: background Services, which are used to execute shorts tasks that are not directly observable or important to the user, and foreground Services, which are supposed to run for a longer time period [35].

Background Services

Background Services are used for performing tasks in the background and not supposed to run for a long time. A background Service may for example write data to a file or download an image from the Internet. Therefore, there are no indications in the UI that a background Service is currently running. The operations performed by a background Service cannot, however, persist after the user has navigated away from the application or the device is put in sleep mode. Starting with API level 26 (Android version 8.0), background services are limited to only run for a short period before being killed by the system [36]. The reason behind this is that Services consume additional resources and keeping them running open may reduce the battery life of the device.

Foreground Services

For tasks that are meant to run for a longer time period, we may use a foreground Service. Unlike a background Service, which is not visible on-screen, a foreground Service must show an indication in the status bar to let the user know that a Service is running and consuming battery. Figure 4.2 shows an indication in the status bar of a foreground Service running in the background when

the application is closed. Consequently, foreground services are not bound to the same restrictions as background services with respect to resource usage. A foreground service may, for instance, be used to playback media in the background that continues playing even when the user has navigated to another application or put the device to sleep mode. Foreground services must request permission to run by being declared in the `AndroidManifest.xml` file.

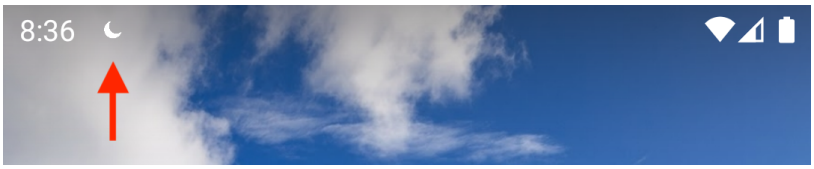


Figure 4.2: Indicator of a foreground service running in the background

Service Lifecycle

A Service is by default not bound to an `Activity`, i.e., it may continue to exist in the background despite the `Activity` it was launched from, is destroyed. In Section 4.2.1, we explained the lifecycle of an `Activity` by describing the different states of an `Activity` and the transition between them. Services follow a similar lifecycle that resembles the Activity lifecycle, but it is far simpler due to the fact that they are not meant to be interacted with. In Figure 4.3, the lifecycle of both an unbounded and a bounded service is shown. When a `Service` is first created, the `onCreate` method is called in which the initial setup is performed. Depending on whether the `Service` is bound or not, different methods are called to handle the start of a `Service`. The `Service` will continue running until all of the bound clients choose to unbind, or if the `Service` is stopped explicitly. In either case, the `onDestroy` method is called to notify that the `Service` is terminated.

4.3 WebSockets

WebSocket is an application layer protocol for enabling real-time bidirectional messages between two machines on the same computer network or over the Internet. The protocol is similar to client-server architectures where one machine acts as the WebSocket server to which one or multiple clients can connect. It is designed to work with existing HTTP-enabled platforms for establishing a connection, but after a connection has been established, messages can be sent over the WebSocket connection [37]. In contrast to standard TCP socket connections

[7] `https://developer.android.com/guide/components/services`

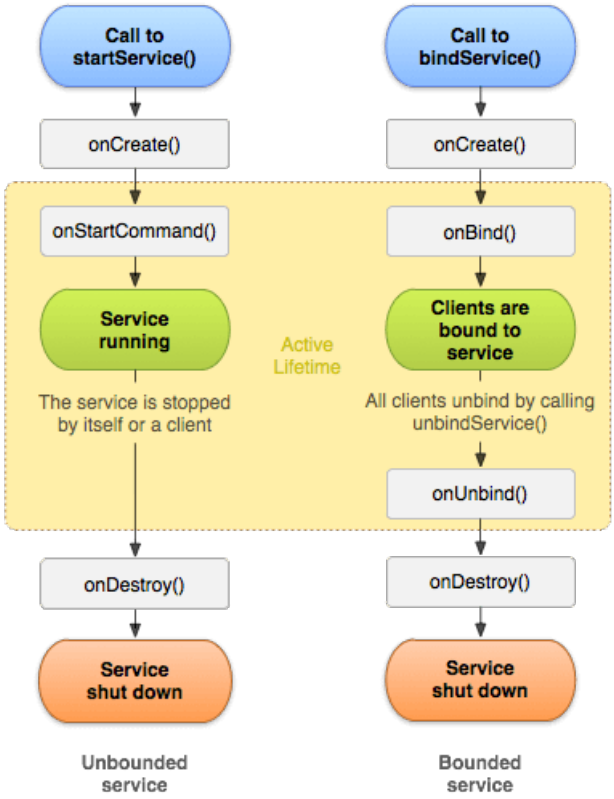


Figure 4.3: The service lifecycle [7]

where two sockets are needed to enable bidirectional communication, WebSockets are designed to send and receive messages over a single TCP socket [37]. This reduces the complexity when multiple clients are connecting to the same server as the number of sockets is half as much as compared to traditional network sockets.

4.3.1 WebSocket Handshake

As can be seen in Figure 4.4, all WebSocket connections start with a handshake between the client and the server. The server is continuously listening on a TCP socket for a connection request from a client and a handshake is initiated when a client sends an HTTP GET request to the server to upgrade the connection. In Listing 4.2, we show an example of a connection upgrade request to the *chat* endpoint of the server at *server.example.com*. An example of a response is shown

[8] `https://www.pubnub.com/blog/websockets-vs-rest-api-understanding-the-difference/`

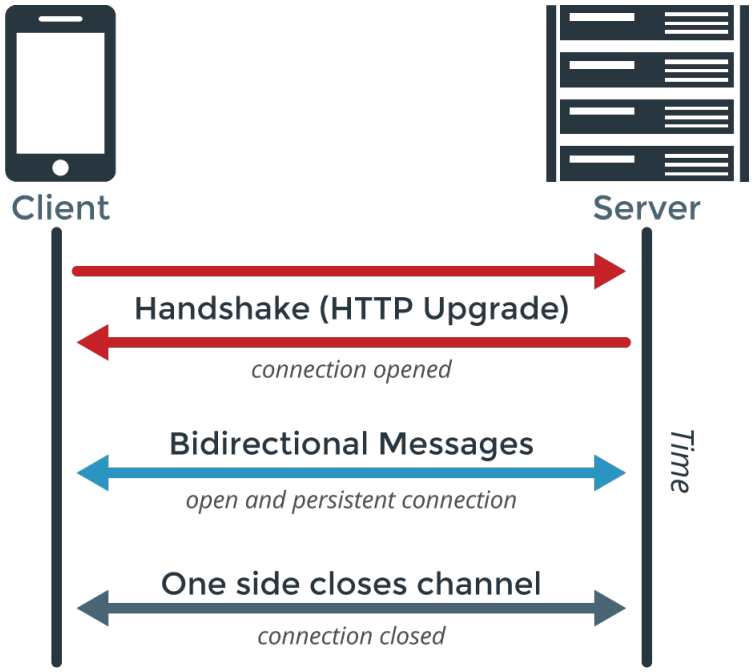


Figure 4.4: A WebSocket connection between a client and a server with initial handshake [8]

in Listing 4.3. Note that this address can be a local device on the same network as the requester, i.e., it does not have to be a remote server. The upgrade request is processed by the server and a corresponding response is sent back. After the handshake, both the client and the server can send messages until the connection is closed by either of the participants. If the client wants to reconnect to the server, a new handshake must take place [37].

Listing 4.2: A request from a client to upgrade the connection to WebSocket.

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6 Origin: http://example.com
7 Sec-WebSocket-Protocol: chat, superchat
8 Sec-WebSocket-Version: 13
```

Listing 4.3: Response from a server to connection upgrade request.

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
```

```
3 Connection: Upgrade
4 Sec−WebSocket−Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5 Sec−WebSocket−Protocol: chat
```

4.4 Databases in Android

After investigating the available database options for Android, we have decided to examine two of them, specifically, SQLite and Realm. SQLite is an ordinary relational database and is preinstalled in Android. Realm is a non-relational database targeted mainly for smartphones [38].

4.4.1 Relational Database

A relational database persists data in entities that are related to one another. Each entity in a relational database usually represents a data model in an application which consists of attributes in columns and data entries in rows. The relations between the models of the application are mapped accordingly in the database with the use of primary and foreign keys. The main advantage of a relational database over alternative databases is the normalization of the data in multiple entities which can later be used to construct novel combinations of the tables through *join* operations. A query language, e.g., SQL, is often used for creating, altering, and querying relational databases.

SQLite

SQLite is an open-source, zero-configuration database engine with support for ACID transactions [39]. Implemented in the C language, the source code of SQLite and the data managed by the database engine are stored in a single file. However, unlike enterprise database solutions, SQLite databases do not support concurrency and distribution as the data is stored in one file. Therefore, it is more suitable for applications without concurrency and consistency needs.

4.4.2 Non-relational Database (NoSQL)

A non-relational database, often referred to as a NoSQL database, is an alternative and novel database paradigm which has gained popularity in recent years. With the advent of big data, conventional databases are not performant nor flexible enough to handle the data load. Unlike traditional relational databases where data is stored using strictly defined schemas, a NoSQL database is more

flexible since data must not be stored in structural tables. There exist no strict rules for how data should be stores but typical approaches are key-value stores, object stores, and graphs [40].

Realm

Realm is a cross-platform, non-relational database for mainly mobile operating systems, e.g., Android or iOS, but is available in multiple programming languages [38]. Realm stores objects created directly on disk instead of mapping them to relations like traditional database solutions. Therefore, Realm databases are able to perform much faster in both read and write operations.

4.4.3 Comparison of Database Alternatives

Databases in Android applications may be used to store large volumes of health data from smartwatches or sensors and must therefore be evaluated thoroughly. In the literature, various database solutions for Android have been benchmarked to evaluate the read and write performance. In [41], an architecture for collecting physiological data from sensors is designed by benchmarking several database alternatives for Android. The results demonstrated that NoSQL databases were performing similarly with respect to write performance. However, the read performance was approximately 140 times faster using NoSQL databases over relational databases.

Similar results have also been reported in [42], where SQLite, a relational database, is benchmarked against non-relational databases like SnappyDB and Realm. The paper evaluates the performance of the databases with respect to read and write operations in Android. All database alternatives perform similarly when writing smaller amounts of data. However, when writing large amounts of data, the SQLite database uses significantly more time to insert entries. The results also demonstrate the benefits of NoSQL databases such as Realm and SnappyDB with respect to reading performance. The Realm database performs better than both the built-in SQLite database and SnappyDB. The difference is more clear when the amount of data to be read increases.

Part III

Design and Implementation

Chapter 5

Design

In this chapter, we present the requirements that our smartwatch application and smartphone application should be able to fulfill. We also propose a high-level architecture for enabling sensor data transfer between Hypnos and Nyx and discuss the different aspects related to the data transfer between the devices.

5.1 Requirement Analysis

In this section, we assess the requirements that our application should satisfy. The requirements are formulated with respect to the aims of our research defined in Chapter 1. Therefore, fulfilling the requirements should give an indication of whether or not we have reached the aims of our thesis. In Sections 5.1.1 and 5.1.2, we discuss the non-functional requirements and functional requirements, respectively.

5.1.1 Non-functional Requirements

The non-functional requirements express the qualities of our application rather than specific functionality that must be implemented. For the data acquisition application, we discuss the following non-functional requirements: extensibility, resource efficiency, privacy, and simplicity.

Extensibility

Our application must be extensible such that new hardware sensors can easily be added as additional components, without requiring large changes to the codebase or an understanding of the underlying implementation. As can be seen in Table 5.1, the Ionic smartwatch investigated in our thesis has a wide variety of hardware sensors. The SDK offers APIs for the accelerometer, altimeter, gyroscope, and heart rate sensor. APIs for the pulse oximeter sensor and the temperature sensor are not available at the time of our research. If and when APIs for the remaining sensors are made available, it should be possible to add them to the smartwatch application effortlessly and used for collecting data. For this to be possible, the

application developed for the Android platform must also be developed with extensibility in mind such that data from the new sensors can be stored in the database.

Table 5.1: The sensors of the Fitbit Ionic

Sensor	Status	API Availability
Accelerometer Sensor	Enabled	Available
Heart Rate Sensor	Enabled	Available
Altimeter Sensor	Enabled	Available
Gyroscope Sensor	Enabled	Available
Pulse Oximeter Sensor	Enabled	Unavailable
Temperature Sensor	Disabled	Unavailable

Resource Efficiency

Smartwatches and smartphones are highly mobile devices and thus have a limited amount of battery capacity. The Ionic smartwatch has multiple sensors that continuously track the physical activity and sleep of the user, modems for Bluetooth and GPS connectivity, and a high-resolution LCD display that can reach up to 1000 nits in brightly lit environments [43]. The aforementioned features have a considerable impact on the limited battery life of the Ionic smartwatch. Any application developed for the Ionic smartwatch must therefore consider the functionality with respect to resource usage. Also worth mentioning is that third-party applications are limited to 64 kilobytes [44] of memory, which is comparable to desktop computers from the 1980s. This limitation restricts the development of a complex application. Therefore, the functionality implemented for the smartwatch application must be picked carefully. The internal architecture of our application and handling of the sensor data affects the memory usage which in the worst case may exceed the memory limitations.

Hardware sensors of the Ionic watch consume battery when they are activated. It is therefore recommended that the sensors are disabled when the display is off so that they no longer consume battery [45]. However, our application is designed for collecting sensor data throughout a sleep session and therefore, cannot disable the sensors. Our application for sensor data acquisition must therefore consider the limited resources and avoid functionality that consumes battery.

Privacy

Health-related data is sensitive and particularly important to keep private to the owner of the data. The issue is more relevant when the data is transported to a remote server and stored in a centralized manner. A vulnerability or weakness in the centralized server will impose immense security risks as attackers may take advantage of such flaws, which ultimately puts the health data in the wrong hands. Measures against such attacks may be to encrypt the data with advanced algorithms or anonymize the data by removing any information that can be traced back to the individuals.

Since our application will be used for collecting sensitive data, e.g., heart rate, movement, and potentially oxygen saturation, we must design a system where the data can be transferred and stored in a way ensuring the privacy of the user. There are two possible approaches with regard to the transmission of the data. We may either

1. transfer the sensor data from the smartwatch to a remote web server that is accessible via the Internet and retrieve it from a smartphone application, or
2. transfer the sensor data directly from the smartwatch to the smartphone.

The former approach imposes security and privacy risks as web servers are susceptible to attacks. The latter does not route data via the Internet and, therefore, has a less likelihood of encountering a privacy breach. However, for both approaches, there is a certain risk of man-in-the-middle (MITM) or eavesdropping attacks by exploiting wireless signal flaws. But since such attacks must be carried out in extreme proximities, the possibilities of them taking place are negligible.

Simplicity

The solution developed in this thesis should be simple with respect to carrying out a new session for the acquisition of sensor data and require as little configuration as possible. In an ideal solution, no configuration should be necessary or compulsory to perform a session but rather available as voluntary options. The potential users may not necessarily be involved in the development of the application or examine the source codes afterwards, and therefore unable to configure the applications. Also, the connection between the smartwatch and smartphone application should not require any configuration but rather occur automatically.

The simplicity aspects should also apply to the architectural design such that the sensor data is transferred and stored with minimal overhead. As mentioned in Section 5.1.1, smartwatches have limited battery, memory, and computing resources and the architecture must therefore take into consideration these limitations. A complicated architecture will likely introduce software complexity and thereby software errors that may affect the performance and reliability of the solution. Architectural overhead could also potentially introduce incidental delay which could act as a bottleneck for the system. When designing the architecture, we should therefore opt for the simpler approach to eliminate such errors.

5.1.2 Functional Requirements

In this section, we cover the functional requirement that our applications should be able to meet. As opposed to the non-functional requirements described in the previous section, functional requirements are specific and possible to evaluate with experiments. By evaluating the end result with regard to the functional requirement, we may determine whether the proposed solution fulfills requirements and thus the aims of our research.

First and foremost, our solution should be able to collect sensor data from the various sensors of the Fitbit Ionic watch. The applications developed should enable us to select which sensors we want to collect data from and which ones we wish to omit, i.e., collecting data from a single or multiple sensors should be possible. Furthermore, the solution should offer a mechanism for extending the array of sensors with novel sensors, if and when enabled by Fitbit. The Ionic smartwatch has built-in sensors which are not enabled for testing and development, e.g., oximeter sensor and temperature sensor, which may be enabled at a later point. Therefore, extending the solution with new sensors should be a simple process that does not require substantial changes to the application.

Using the applications, it should be possible to record an overnight session with multiple sensors of the smartwatch at various frequencies. The applications must be designed such that the duration of the sessions does not impact the ability to collect data. It is evident that keeping multiple sensors active throughout an overnight sleep will have an impact on the battery, but the impact should not prevent the use of the smartwatch for typical operations, e.g., record physical activity, deliver notifications, and track exercises. Since the applications may be used for various purposes, functionality for configuring the sensors with regard to frequency and state (enabled or disabled), must be available. Preferably, the configuration should not require changes to the implementation. Instead, the configuration options should be presented within the application menus.

After a session has been recorded, the data should be available within the

smartphone application. It should be able to store multiple sessions regardless of the frequency and the sensors used. In the smartphone application, the sessions should be listed with details regarding the duration, included sensors, and at what frequencies the sensors were configured during the session. Although not crucial for the data collection, we should also implement functionality for visualizing the data using charts. Lastly, since the analysis may not be performed on the smartphone, we should offer a mechanism for exporting the data from the Android application to a computer.

5.1.3 Summary of Requirements

The requirements denoted in the previous sections can be used for evaluating the applications we present in this thesis. In Table 5.2 we list and briefly summarize both the functional and non-functional requirements mentioned in the previous two sections:

Table 5.2: Summary of non-functional and functional requirements.

Non-functional Requirements	
Requirement	Description
Extensibility	The applications must be extensible such that novel sensors may be added with minimal effort. Preferably, changes should only be made to the smartwatch application.
Resource Efficiency	The application developed should take into consideration the limited resources of the smartwatch.
Privacy	Collected data must be handled with user privacy in mind. If data is sent over the Internet, appropriate security measures should be taken.
Simplicity	Recording a session must require few steps and configuration should be easy.
Functional Requirements	
Requirement	Description
Add novel sensors	Extending the smartwatch application with novel sensors should be possible by modification of a dedicated configuration file.
Record Long Sessions	The applications should enable recording of longer sessions lasting at least 8 hours and without any disruptions or failure.

Table 5.2 continued from previous page

Sensor Configuration	The application should allow the user to change the sampling frequency of each sensor independently and enable or disable sensors.
View Recorded Sessions	Recorded sessions should be viewable, within the smartphone application.
Store Sensor Data	Store the collected data in a structured manner, e.g., in a database, such that they can be retrieved later.
Export Sensor Data	The sensor data must be possible to export to a secondary medium, e.g., via Bluetooth.

5.2 High-level Design

When designing the architecture of the data acquisition application, we must take into consideration the capabilities and limitations of the Ionic smartwatch. As previously mentioned in Section 5.1.1, the Fitbit Ionic smartwatch is bound by both battery and memory limitations due to its smaller form factor. Therefore, the architecture must be designed such that the smartwatch is able to collect sensor data continuously without disruptions, e.g., running out of memory or battery.

5.2.1 Platform Limitations

Initially, we were aiming to store the sensor data on the smartwatch temporarily during the recording of a session and then transfer the data to the smartphone for permanent storage. A key advantage of this approach is that data can be collected and stored directly on disk without continuously offloading to an external location, e.g., a web server, and thereby avoiding signal disruptions and delays while recording. When a session is finished, sensor data from all sensors can be exported to an external database device for further analysis. The Ionic smartwatch has an internal storage of 2.5 gigabytes [43] which is sufficient to store multiple sessions. However, after further investigation, we discovered that third-party applications are not able to freely utilize the internal storage and are limited to 15 megabytes. This limitation applies to the total storage space taken by an application, including source code, resources, and application data. Fitbit does not disclose why this limitation exists, but we believe it is there to restrict the scope of applications. With this limitation, we cannot store data for longer time periods on device and offload after the session is ended.

Since the collected data cannot be stored on the device itself, it has to be

offloaded to a secondary device. In our case, we intend to create an application for the Android platform which can receive the collected data for permanent storage. In other words, that data must be transferred throughout the recording of a session from the smartwatch to the connected smartphone. Although the Ionic smartwatch has built-in WiFi, is not available to third-party developers. Therefore, data must be transferred via a Bluetooth connection. The Ionic smartwatch uses Bluetooth for synchronizing health metrics and records, e.g., steps, calories, etc., with the Fitbit smartphone application and synchronizes ever so often in the background. When the devices are in proximity, data is regularly pushed to the mobile application and uploaded to Fitbit servers for permanent storage. Bluetooth is also used for connecting audio accessories such as wireless headphones and speakers. However, an API for using the onboard Bluetooth for application development is not available. Therefore, routing the sensor data to a smartphone is a great challenge that must be tackled in order to use Fitbit smartwatches for sensor data acquisition in OSA detection.

In [46], Schellevis demonstrates how to reverse engineer the Bluetooth communication between the FMA and the Fitbit HR fitness tracker band in order to obtain the data without being routed via Fitbit servers. A backdoor, which has later removed, is exploited in order to access encryption keys used to decrypt the Bluetooth traffic between the smartwatch and companion phone. Such an approach is deemed not viable for our application because a software patch would disable our application from working with future versions of the firmware. Furthermore, exploiting and documenting a software backdoor is going to impose security concerns for the platform since it may be used for malicious intents. Most of the data collected and transferred by the smartwatch itself is processed and does not reflect our needs with respect to raw sensor data. Therefore, we should opt for a solution that is able to fetch raw sensor readings.

5.2.2 Potential Architectures

Applications developed for Fitbit smartwatches may have a companion application that is operating within the FMA on the smartphone. As mentioned in Section 3.2, the companion is able to augment the functionality of the smartwatch application by taking advantage of the connected smartphone. The Fitbit SDK offers two APIs for communication between the Ionic and the companion application, that is, the Messaging and File-transfer API. These APIs may be used for transferring collected sensor data from the smartwatch application and thereby eliminate the need for a Bluetooth API. In Sections 5.4.2 and 5.4.3, we discuss the capabilities and limitations of both APIs for sending sensor data.

Using these APIs, we are able to transfer the data to the companion appli-

cation. At this point, the data is already located on the smartphone, but within the FMA. Due to the application sandboxing feature of the Android operating system [47], data of an application cannot be modified or read from another application without explicitly receiving permission. Consequently, there exists no procedure for accessing the Fitbit mobile data directly from the file system. We believe this is a deliberate design choice by Fitbit to prevent malicious applications from accessing application data that may contain sensitive information. Since we were not able to access the data directly, we evaluated other approaches for transferring the data to our Android application. Limited by the Fitbit SDK, we evaluated mainly two approaches for transferring data from the companion app to our Android application:

1. transfer the sensor data to a remote server using the Fetch API available for the companion application from which the Android application may pull data, or

2. create a local WebSocket server within the Android application and communicate with the server using the WebSocket API available through Companion API.

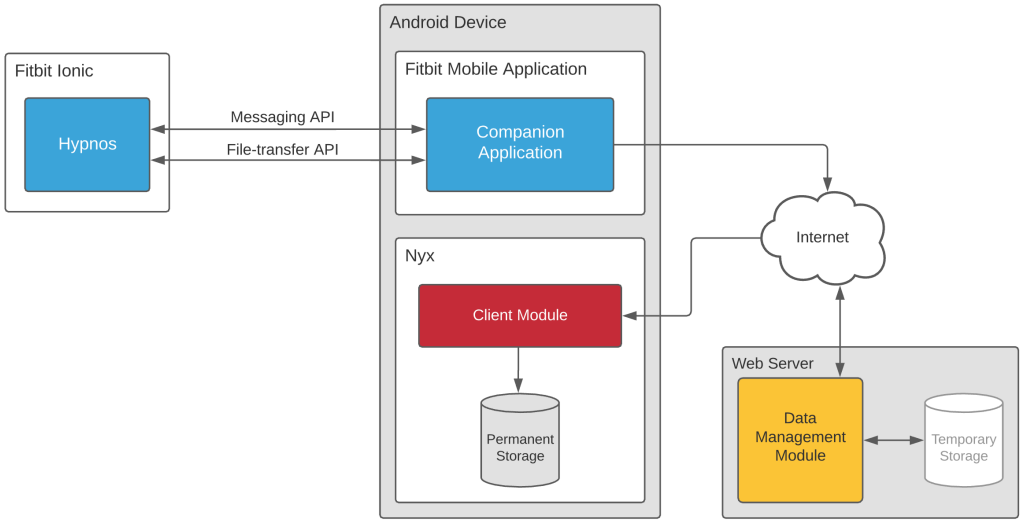


Figure 5.1: Web server architecture which routes the sensor data via the Internet.

The former approach, which is illustrated in Figure 5.1, requires that we route the sensor data, in real-time, through a web server for temporary storage. Further, the Android application should retrieve this data for permanent storage.

This approach introduces additional complexity to the data transfer pipeline as the data has to be sent to the companion application which further sends it to the web server and finally retrieved from the Android application. As discussed in Section 5.1.1, routing the data through the Internet introduces security implications, but also delay. Data may be encrypted and decrypted before being sent to the server and after retrieval, but this introduces an overhead to the system. Furthermore, a web server may be susceptible to attacks without proper and continuous security measures.

The latter approach requires a WebSocket server that runs locally on the Android smartphone and can receive messages sent from a client. WebSocket is a protocol for bidirectional communication between a server and one or multiple endpoints. In Section 4.3, we explain the WebSocket protocol in detail and how it differs from conventional network socket connections. Since the Companion application supports the JavaScript WebSocket client API, we may establish a connection between the companion and the server within theAndroid application. This connection may then be used to send sensor data from the companion directly, without being routed via a web server.

5.2.3 Architectural Overview

In our final architectural design, we propose a solution where a connection between the companion application and the Android application is establishing using the WebSocket protocol. In contrast to the web server architecture discussed in Section 5.2.2, the proposed solution avoids routing data via the Internet and eliminates the issues that arise therein. In Figure 5.2, a high-level architecture is illustrated by showing the communication between the applications. As can be seen from the figure, the companion application acts as a mediator between the smartwatch application and smartphone application, which otherwise are unable to communicate locally. The final architecture is dependent on the Messaging and File-transfer API for communication between Hypnos and companion and a WebSocket connection between the companion and Nyx.

5.2.4 Application Roles

In our final architecture suggested in the previous section, it is required that we develop three applications to achieve the aims of this thesis. For the smartwatch, we develop an application called Hypnos. Although it is mainly targeted at the Fitbit Ionic, it should run on different smartwatches from Fitbit that use the same version of Fitbit OS as the Ionic smartwatch. For the Android platform, we develop an application called Nyx that is intended to be run on most versions

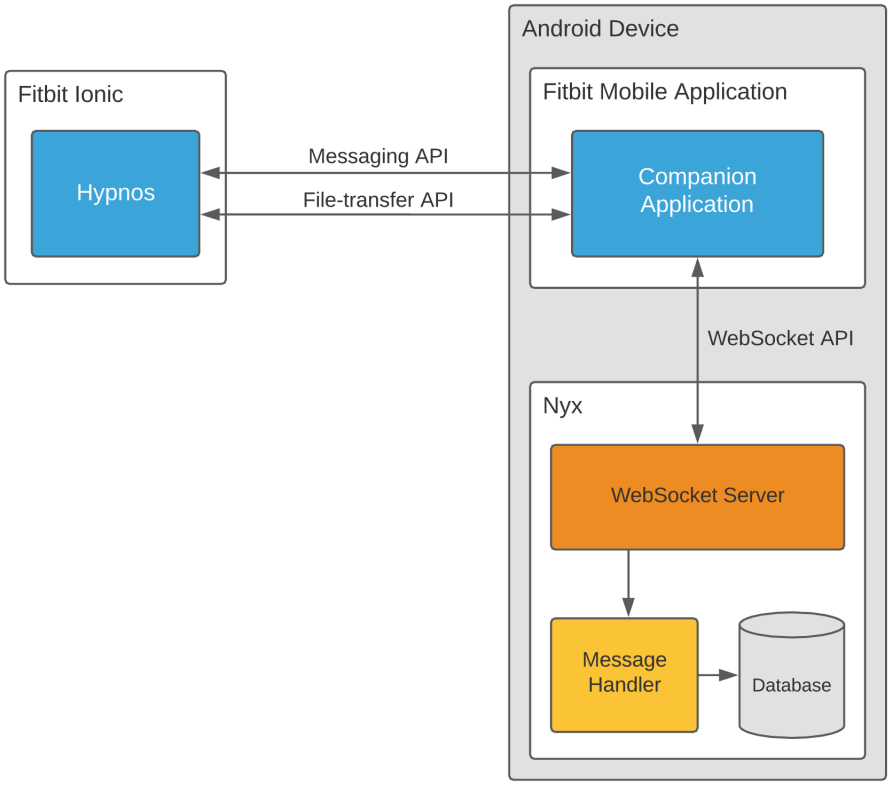


Figure 5.2: High-level architecture of our solution.

of Android. The applications and their specific roles are as follows:

Hypnos

Hypnos is mainly responsible for controlling the hardware sensors and collecting data for dispatch to Nyx. From Hypnos, we should be able to control the lifecycle of a session by initiating, starting, and stopping. Some sensors output data that barely change. Therefore, Hypnos should allow us to change the sampling rate of each sensor independently to allow for fine-grained control. Toggling on and off sensors should also be an option to save battery if a sensor is not needed.

Companion

The companion is not a standalone application but an extension of Hypnos running within the FMA. It serves as a bridge between Hypnos and Nyx by commu-

nicating with both endpoints. Although it is referred to as a separate application, in practice, the companion and Nyx run within the same Android device. This is necessary for the transmission of messages from Hypnos to Nyx, which would not be possible without.

Nyx

Nyx is supposed to run on Android devices and act as the persistent storage component. The application should house a database solution that is able to store sensor data which can be retrieved later for previewing or exporting to a computer. The purpose of Nyx is not to process the data in any way. Our aim for Nyx is able to store data from novel sensors without any modification to the application source code.

5.3 Communication Mechanism

The Fitbit application must be able to communicate with the Android application wirelessly in order to transmit sensor data. At the time of our investigation and development, a Bluetooth API was not directly available but accessible via APIs that abstract away the lower layers (link and physical layers) of the protocol. For transmitting data via Bluetooth, only application layer APIs are available, e.g., Messaging and File-transfer API. These APIs simplify the development process since we do not need to scan for Bluetooth devices or establish a connection manually. Messages that contain sensor data may therefore be forwarded using either the Messaging or File-transfer API or a combination of both. Data transferred using these APIs are received at the companion within the FMA. To send messages between Hypnos, the companion, and Nyx, we need a format that defines the possible structure of the messages.

5.3.1 Message Structure Design

Initializing a new connection and transmitting sensor data requires a predefined message format that all applications in our solution are able to comprehend. The messages are defined in JSON since it allows fine-grained structures. Listing 5.1 contains the message format used to allow communication between Hypnos, the companion, and Nyx. The `command` field defines the purpose of the message and the appropriate action that should be taken upon reception. The `payload` defines additional parameters regarding the command being sent.

Listing 5.1: Message format used between Hypnos and Nyx.

```
{
    command: "COMMAND_TO_SEND",
    payload: {
        ...
    }
}
```

5.3.2 Communication Protocol Design

The communication between Hypnos and Nyx must be established before any messages can be transferred. Since a session should be started and stopped from the smartwatch, the initial connection procedure must be actuated from Hypnos. When Hypnos is launched, a command should be sent to the companion that initiates the connection to the WebSocket server running within Nyx. We define a command `START_SEARCH` which notifies the companion that Hypnos is ready and that it should start looking for Nyx on the Android device. In Listing 5.2, an example of this command is shown. Along with the message, the model identifier of the Fitbit smartwatch is sent. Since we are dealing with the Fitbit Ionic, the `modelName` field of the message is "Ionic". For other Fitbit smartwatches, e.g., Versa or Sense, this identifier should be different.

Listing 5.2: The command sent from Hypnos to the companion for establishing a connection.

```
{
    command: "START_SEARCH",
    payload: {
        modelName: device.modelName,
        battery: battery.chargeLevel
    }
}
```

A WebSocket connection, once established, is active until either the client or the server chooses to close the connection. After a successful connection, we therefore do not need to perform any additional actions. Following a successful connection, the companion should notify Hypnos that Nyx is available and ready to store data from the sensors. Therefore, a message is sent containing the command `CONNECT` from the companion to Hypnos. Only when this command is received, Hypnos should allow recording a new session. If the connection to Nyx is lost at any point, a command `DISCONNECT` should be sent to Hypnos.

Before a data collection session is started, Hypnos should send a command to notify Nyx of the configuration of sensors within Hypnos, e.g., which sensors

Table 5.3: Communication protocol between the applications

Command	**Sender**	**Recipient**	**Description**
START_SEARCH	Hypnos	Companion	Initiates a search for the WebSocket server.
CONNECT	Companion	Hypnos	Informs Hypnos that Nyx is available.
DISCONNECT	Companion	Hypnos	Informs Hypnos that Nyx is unavailable.
INIT_SESSION	Hypnos	Nyx	Sends sensor configuration, i.e., state and sample rate.
START_SESSION	Hypnos	Nyx	Starts a new data collection session.
ADD_READING	Companion	Nyx	Add new readings to the database.
STOP_SESSION	Hypnos	Nyx	Ends the data collection session.

are enabled and the frequencies of the sensors. Also, a unique session identifier should be generated and sent to Nyx. The unique identifier should be sent along with all subsequent messages. This way, when we send sensor data from Hypnos, Nyx is able to insert the readings to the correct session. For this purpose, we define the command `INIT_SESSION`. Furthermore, we must define commands which notify Nyx that a session is started and ended. These commands should contain the timestamps such that we are able to register the duration within Nyx. At the start of a new session and after it has ended, we should therefore send the commands `START_SESSION` and `STOP_SESSION`, respectively. During a recording session, readings are first sent to the companion from Hypnos. From the companion, they should be forwarded to Nyx via the `ADD_READING` command. In Table 5.3, we summarize the commands of the communication protocol. Notice that not all messages should be directly relayed to Nyx, e.g., `START_SEARCH`, since the connection to

5.4 Data Transfer

In this section, we cover aspects related to the data transfer between the smartwatch application and the companion within the smartphone. First, in Section 5.4.1 we discuss various file formats for transferring the sensor data. Then, we evaluate the available APIs for transferring data and messages in Sections 5.4.2 and 5.4.3.

5.4.1 Data Format Alternatives

During the development of Hypnos, we evaluated various formats for the sensor data transfer. The formats considered in this thesis are the formats that are available for the Ionic smartwatch, that is, CSV, JSON, and binary files. In this section, we briefly evaluate these formats with regard to data density, structure, and efficiency.

CSV

CSV, or Comma-separated values, is a data format standard that consists of rows with data records whose values are separated by a comma. Although the name implies a comma, other characters, e.g., semi-colon or underscore, can be used instead for separating the values. The CSV format has been evaluated during the design of Hypnos. Listing 5.3 shows an example of how readings from a heart rate sensor may be structured. As can be seen, we organized the readings as rows such that the columns are fixed, but the rows are variable. In Nyx, the data could then be parsed row-by-row. However, a clear disadvantage with this approach is the duplication of the values for each reading.

Listing 5.3: Sensor data formatted in CSV.

```
1 session_id, sensor, timestamp, data
2 906ca3d3−abd3−48d7−a323−86a42e3b7809, HEARTRATE, 1617914698074, 70
3 906ca3d3−abd3−48d7−a323−86a42e3b7809, HEARTRATE, 1617914699034, 72
4 906ca3d3−abd3−48d7−a323−86a42e3b7809, HEARTRATE, 1617914700054, 71
5 ...
6 906ca3d3−abd3−48d7−a323−86a42e3b7809, HEARTRATE, 1617914701063, 68
```

JSON

JSON is a standard for object transfer that is used for transferring data between the user applications and backend infrastructure. The standard, which is based on data types of JavaScript, consists of primitives such as numbers, booleans, strings, and null and collections such as arrays and dictionaries [48]. In our smartwatch application, we have attempted to use JavaScript objects for structuring sensor data before it is sent to the companion and Nyx. An example of this format is shown in Listing 5.4. Since JSON is based on JavaScript objects, we do not need to convert the sensor readings from the objects to JSON. However, a disadvantage of using JSON is the descriptions which take up extra space

which increases the size of the data being sent.

Listing 5.4: Sensor readings as JSON

```
{
  "command": "ADD_READING",
  "payload": {
    "sessionIdentifier": "6a9815e6-b2ba-4fe9-a371-331d45fe4f15",
    "sensorIdentifier": "HEARTRATE",
    "timestamps": [
      1617914698074,
      1617914699034,
      1617914700054
    ],
    "data": [
      {
        "type": "BPM",
        "items": [
          70,
          71,
          72
        ]
      }
    ]
  }
}
```

Binary

Using the Fitbit SDK, we may store binary files on the file system. A key advantage of this format is the storage efficiency as there are no data descriptions occupying additional space. As a result, the total size of the data is smaller, which should enable faster data transfer from the smartwatch to the connected smartphone. However, persisting binary files is complicated since it requires that we allocate the exact number of bytes needed to store the sensor data.

5.4.2 Messaging API

The Messaging API allows sending messages from the smartwatch to the companion and may therefore be utilized to transfer sensor data. Using this API, we may also send commands related to the recording of a session which are described in Section 5.3.2. The maximum amount of data that can be sent in one message using the Messaging API is 1024 bytes [49]. Depending on the size of a sensor reading, this limitation imposes restrictions on the number of readings that can be transferred in a single message. Using the format shown in Listing 5.4, we may send approximately 20 readings from the heart rate sensor. The simplest

approach to increase the number of readings per message is through *minification*, i.e., remove the empty characters that count towards the message limit. Characters such as line breaks, tabs, and spaces increase readability for humans but are strictly not necessary. also the size of the message. Excluding white spaces from the message yields nearly 40 heart rate readings. However, for sensors that have multiple properties, e.g., accelerometer or gyroscope, we would not be able to fit as many readings in one message.

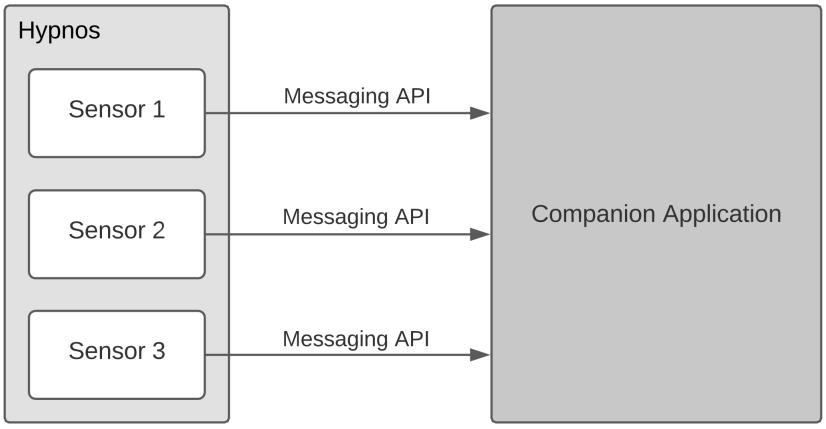


Figure 5.3: Initial architecture for data transfer

In our first architecture for Hypnos, each sensor collected and dispatched its readings to the companion using the Messaging API. In other words, since multiple sensors are enabled during a session, Hypnos is sending one message for each sensor. In Figure 5.3, we illustrate the transfer of sensor data using the Messaging API. For one or two sensors this architecture is working as intended. However, if the number of active sensors increases or the frequency of the sensors are above 1 Hz, the Messaging API is not able to relay the readings to Nyx for persistent storage. It is possible to batch multiple readings into one message, but we are still limited to a maximum of 1024 bytes. For shorter sessions lasting one or two hours, the Messaging API is able to handle approximately 5 messages per second. However, when the duration of a session reaches 2 to 3 hours, no messages are being transferred as the capacity of the messaging channel is exceeded. Therefore, this API is not reliable enough for sending data from multiple sensors at higher frequencies during an overnight sleep session. Our understanding is that this communication channel is more suited to smaller messages sent less frequently. We may therefore send the commands using the Messaging API.

5.4.3 File-transfer API

The File-transfer API can be used for sending files bi-directionally between the smartwatch application and the companion. It is not required that a file is persisted to disk before being sent and may therefore reside in memory. Unlike the Messaging API, which has a limit of 1024 bytes, the File-transfer API has no size limit. In theory, we should be able to send arbitrarily large chunks of sensor data from the smartwatch to the companion. In our testing, we experimented with the File-transfer API to send data from each sensor directly to the companion. However, sending multiple files results in erroneous behavior as the transmission speed is lower than for the Messaging API. Whether the issue is caused by the File-transfer API or the underlying Bluetooth module of our test device, has not been investigated. To reduce the number of files being sent with the File-transfer API, we implement a mechanism for batching the data. In Section 5.5, we explain various mechanisms for batching which have been evaluated throughout the development of Hypnos.

5.5 Batched Dispatch

Since our solution must be able to record a sleep session lasting at least 8 hours with multiple sensors, we should limit the number of data transfers to the companion application. By accumulating the readings in batches we may dispatch sensor data at a lower frequency and thereby limit the number of files being sent. In theory, lowering the file transfer rate should allow for the companion to handle each file. However, sending data less frequently means that Hypnos must store data temporarily instead of dispatching it right away. As mentioned earlier, the JavaScript runtime and heap memory of Fitbit applications are limited to 64 kilobytes. A large fragment of the memory is occupied at all times since the JavaScript classes of our application also count towards this limit. When the application is in an idle state, approximately 60% to 70% of the memory is in use. Caching data locally for longer periods may therefore easily exceed the memory and crash the application immediately. To avoid running out of memory, we evaluated mainly two techniques to batch the sensor readings:

- **Time-based Dispatch:** Dispatch readings based on a fixed interval and disregard the number of collected readings, e.g., send data every 30 seconds. The more readings that are to be sent to the companion, the more memory is used by the sensor data.
- **Count-based Dispatch:** The alternative method for dispatching messages is by keeping track of the number of readings that are awaiting and send the data when the batch has reached a predefined number of events.

The time-based approach is naive because the number of readings may vary greatly depending on the number of sensors that are active and the frequency at which they are configured. Using multiple sensors at higher frequencies may therefore accumulate more readings than sent and exceed the available memory. On the other hand, using the count-based approach we can be sure that the will not be any issues regarding the memory since we explicitly declare a limit. Regardless of the dispatch method, we should store the data temporarily in memory to limit the rate of data transfer.

5.5.1 Memory Cache

Initially, we cached sensor data in memory by defining a data structure to which all sensors appended readings. For lower frequencies and few sensors, this architecture works as intended. The new architecture with the memory cache is shown in Figure 5.4. However, once we use three to four sensors and increase the accumulated frequency to about 10 Hz, we reach the upper limits of the memory. The application is only able to collect data for approximately an hour before running out of memory and crashing during the data collection. To avoid running out of memory, we attempted to send messages more frequently, but this resulted in the companion application closing. Due to restrictions with the File-transfer API described in Section 5.4.3, sending files frequently is not feasible as it will result in the transfers failing. We experimented with various sizes for the memory cache and frequency of dispatch, but this architecture is not reliable enough to record longer sessions.

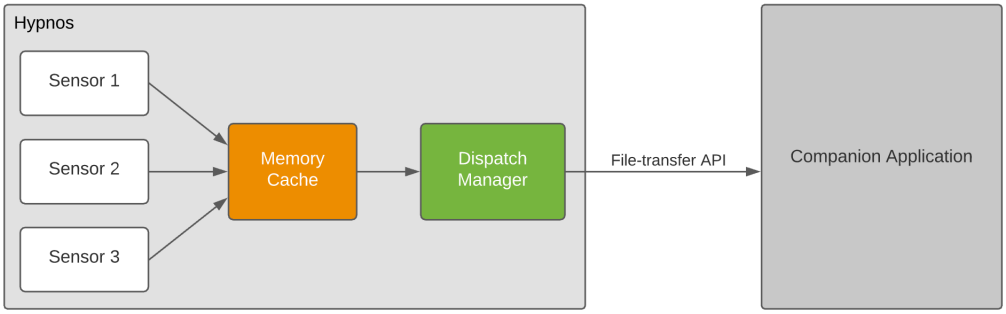


Figure 5.4: Memory cache used to reduce the frequency of dispatch of sensor data

5.5.2 Memory and Disk Cache

Since we cannot store more than a certain amount of sensor data in memory nor send the data frequently, we attempt to offload sensor data from the memory to disk. As illustrated in Figure 5.5, the disk cache is used to persist the readings in the memory to disk in order to avoid overloading the memory limitation of 64 kilobytes. The File-system API, which is explained in detail in Section 3.3.1, allows writing files to the internal storage of the smartwatch. Although Fitbit applications are limited to 15 megabytes, we may persist files temporarily to avoid using a large memory cache. Initially, files containing sensor data were stored as JSON files, as they allow building hierarchical structures. When the number of readings reached a certain threshold, a new file was created containing the readings in memory. Although it is technically possible to store more than one batch in the same file, since a JSON file must be loaded into memory to append new readings, it is not possible to write multiple batches in a single file. Therefore, a new file is created each time the memory cache is full.

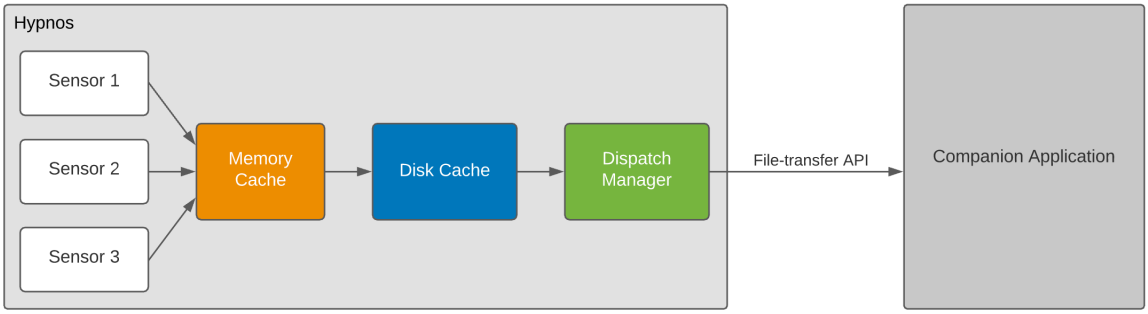


Figure 5.5: Disk cache used to avoid memory overload during data collection.

Using this architecture, we were not able to collect data for more than approximately 2 hours. During the preliminary testing, we noticed that the memory usage of Hypnos was gradually increasing until exceeding the memory limit and crashing. After investigating this issue further, we suspect that writing JSON files to disk continuously is causing a memory leak, which ultimately crashes the application after a while. The memory usage was logged throughout the session using dedicated APIs offered by the Fitbit SDK.

5.5.3 Disk Cache

During the experimentation with the memory and disk cache, we realized that relying on the memory to batch sensor readings, will eventually exceed the lim-

itation of the Ionic smartwatch. At the same time, we cannot dispatch data more frequently since the APIs for data transfer are not able to handle the increased load. In order to avoid running out of memory, we attempt to remove the memory cache and avoid using the memory to accumulate readings. Instead, the sensor data was written directly to the disk. In theory, since the memory is not used for batching data, we should not overload the memory regardless of the duration of the data collection, the number of sensors that are enabled, or the frequencies they are configured at. However, since the files on the disk must be loaded into memory before new readings can be added, we cannot write multiple batches of data from the sensors in a single JSON file.

In our final architecture, which is shown in Figure 5.6, we accumulate sensor readings in a disk cache which stores the data in binary files. Unlike JSON files, binary files can be opened in *append mode*, which allows us to write more than one batch of sensor data in the same file without loading the file into memory. Therefore we avoid exceeding the memory limit as storing files in a binary format is dependent on the available disk storage rather than memory. On the one hand, by using binary files, we are able to reduce the number of files to be dispatched since we may combine several batches into one file. On the other hand, constructing a binary file is complicated since the exact number of bytes needed for the readings must be allocated in advance. Initial testing using a disk cache with binary files, we observed that we are able to collect data using multiple sensors for at least 12 hours. This indicates that our final architecture is only dependent on the battery life, as the memory is no longer an issue.

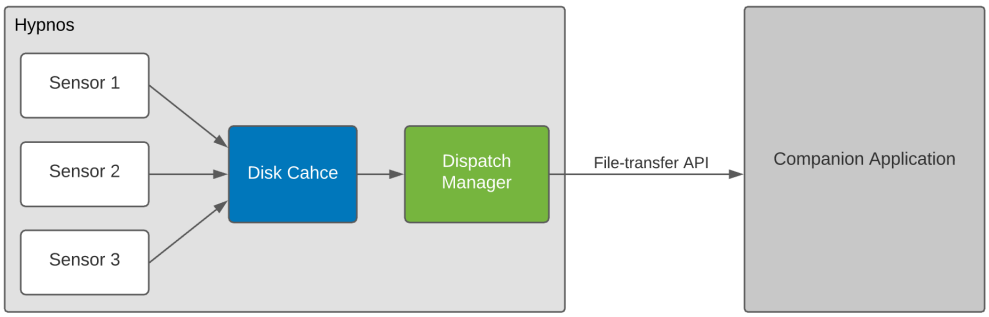


Figure 5.6: Final architecture using a disk cache.

5.6 Delay and Disconnection Tolerance

The collected sensor data is first transferred via Bluetooth to the companion and then to Nyx via a WebSocket connection. Therefore, the proposed architecture

has two areas of potential delay or disconnection that we should account for. In case of a disconnection between the Ionic smartwatch and the Android device, the sensor data should be buffered until the connection is reestablished.

5.6.1 Buffering in Hypnos

If the Bluetooth connection is temporarily lost, neither the Messaging nor File-transfer API can be used to dispatch sensor data. In such cases, the data must be buffered until the connection is reestablished. In our final architecture using a disk cache, we store the sensor data in binary files. At certain intervals, Hypnos attempts to dispatch the files to the companion. If the file transfer is successful, the file is deleted from the disk to avoid reaching the storage limit. However, if the connection is lost, the files cannot be transferred. Therefore, we attempt to send the buffered files again at the next dispatch.

5.6.2 Buffering in the Companion

Normally, when the companion receives data from Hypnos, it should be directed to Nyx via the WebSocket connection. However, if the Android application has stopped or relaunched due to an error, readings cannot be forwarded. Therefore, the data received during this period will be lost unless it is handled by the companion. To avoid this, we store the readings from Hypnos temporarily on the companion. When Nyx is available and a WebSocket connection is reestablished, the accumulated readings are sent.

5.7 Fitbit Application Design

In this section, we describe in detail, the design of our smartwatch application Hypnos with regard to the UI and application flow. In our requirement analysis, we determined that the sensor configuration with respect to the enabled sensors and their frequencies should be configurable. Therefore the UI of Hypnos must be designed to allow navigating between multiple screens.

5.7.1 User Interface

Since our application should have multiple screens, e.g., when recording a new session, changing the frequency of sensors, and enabling or disabling sensors, we must implement a multi-view application with navigation. The Fitbit SDK does, however, not offer a framework for creating an application with multiple views. During the design of Hypnos, we experimented with multiple third-party

frameworks that implement elementary view navigation. Of particular interest are the frameworks sdk-multi-view [50] and ionic-views [51].

sdk-multi-view

The sdk-multi-view framework defines an array of views and corresponding JavaScript files in order to load views on demand. In other words, when the application launches, only the first view is loaded into memory. When the user navigates to another view, the corresponding JavaScript and SVG file is loaded *lazily*. In Listing 5.5, three views are defined and the first is loaded initially using the `navigate` function. The JavaScript files are loaded lazily using the `import` method. On a device with limited memory, lazy-loading views reduces memory usage considerably. However, loading a new view will remove any references to the previously loaded view and its associated states. Therefore, using lazy-loading we are not able to keep track of a global state within the application, as loading a new view will erase references of the previously loaded view.

Listing 5.5: A variable hold the references to the views of the application.

```
1  const views = init(
2    [
3      ["view-1", () => import("./views/view-1")],
4      ["view-2", () => import("./views/view-2")],
5      ["view-3", () => import("./views/view-3")]
6    ],
7    "./resources/views/"
8  );
9
10 views.navigate("view-1");
```

ionic-views

The `ionic-views` framework is a more sophisticated since it defines each screen in the user interface as JavaScript classes. The class defining the screens must extend the abstract class `View` so that methods are executed according to the navigation between the screens. For example, each time a view is navigated to, a handler method `onMount` is called so that code can be executed. Further, the `onRender` method is executed when a view launches but also when the device wakes from sleep. When the user navigates away from the view, the `onUnmount` method is called. Listing 5.6 shows the bare minimum class for a view with the `onMount` and `onUnmoun` methods. Handling view navigation using `onMount`, `onRender` and `onUnmount` simplifies the application design significantly. In addition to these methods, we extended the framework with an additional

`onPropsChange` method which is called whenever the global state is changed, e.g., when Nyx is no longer available. The `props` parameter is also added to the `onMount` method such that views can be initialized based on the global state.

Listing 5.6: A class representing a view in the user interface.

```
export class Main extends View {

    onMount(props){
    }

    onPropChange(props) {
    }

    onRender(){
    }

    onUnmount(){
    }

}
```

A disadvantage of the `ionic-views` framework is that all views are loaded into memory at launch. Unlike `sdk-multi-view`, views are not lazily loaded as the views are navigated. Therefore, having multiple views will use more memory even if only one screen is shown at a time. Prior to optimizations, memory usage at launch reached up to around 85%, despite only one view being visible on screen. In our first architecture which temporarily cached readings from sensors in memory, the use of `ionic-views` caused memory overloaded errors, which inevitably crashed the application in the midst of a session. To avoid exceeding the memory, we optimized the application, e.g., by only creating memory references that are needed. After such optimizations were done, memory usage in the idle state was lowered to around 65%. Additionally, after caching data on disk instead of memory, we were able to avoid reaching the memory limit altogether.

5.7.2 Application Flow

The initially launched screen should present the main navigation options. From this screen, it should be possible to start a new session or enter the preferences to change sensor configuration. After a session has been recorded, a summary of the session, e.g., reading count, should be presented. Furthermore, the preferences should present options for enabling and disabling each sensor individually as well as changing the frequency. Figure 5.7 illustrates our initial application view hierarchy and the navigation between. As can be seen from the figure, `Main View` is the root of the view hierarchy and there are three options for navigation. The

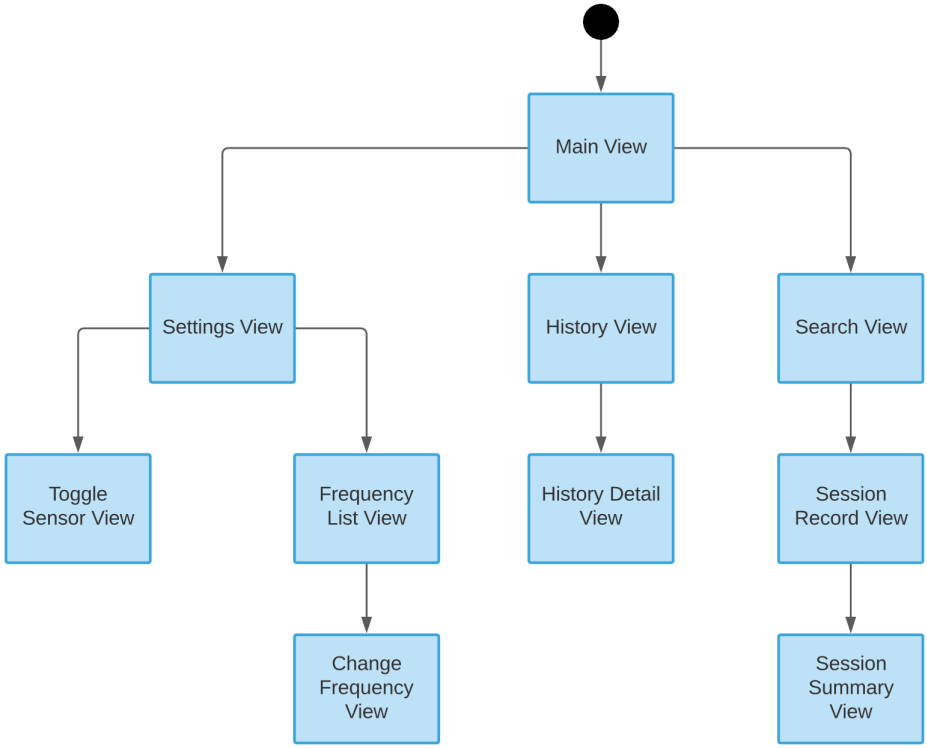


Figure 5.7: Flow chart of our initial hierarchy of views in Hypnos.

first option, `Search View`, initiates a new session by first searching for Nyx. The second option, `History View`, shows the previously recorded sessions and the possibility to view details. The last option, `Settings View`, enters the settings of Hypnos.

Our initial design assumed that sensor data could be stored locally on the device itself throughout a session. In earlier prototypes of Hypnos, we therefore presented the recorded sessions in a list that could later be transferred selectively. However, as stated earlier in Section 3.3.1, it is not possible to exceed the storage limitations of 15 megabytes for third-party applications. Therefore, storing whole sessions locally in Hypnos and selectively sending them to Nyx is not feasible. Consequently, in our updated application view hierarchy, we do not present any information regarding the recorded sessions within Hypnos. The new view hierarchy is shown in Figure 5.8. Notice that there are only two possible options for navigation from the initial view instead of three.

In our first view hierarchy, the steps for recording a session comprised of navigating to a `Search View` that, when a connection to Nyx was established,

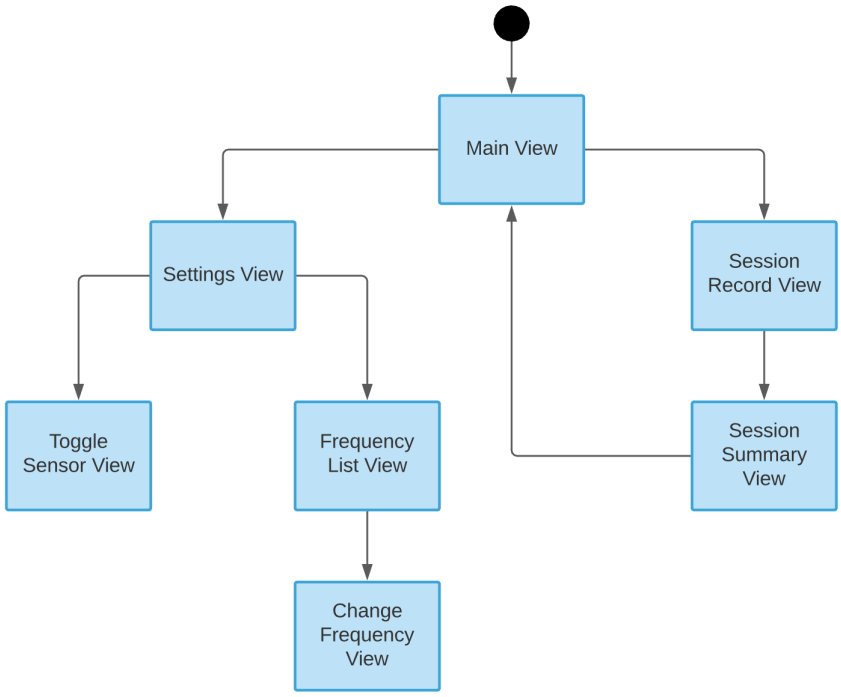


Figure 5.8: Flow chart of our updated hierarchy of views in Hypnos.

redirected to the `Session Record View`. As a part of optimizing the memory usage, we removed the additional `Search View` and instead merged it with the `Main View`. Hypnos could therefore look for Nyx immediately after being launched while saving memory at the same time. Further, the first approach does not consider the possibility of recording multiple sessions without relaunching the application. In our updated view hierarchy, after a session has been recorded, the `Main View` is loaded such that a new session may be started. In Figure 5.8, this is illustrated with an arrow going from `Session Summary View` to `Main View`.

5.8 Companion Application Design

Since Hypnos is not able to communicate with Nyx directly over Bluetooth, the companion application functions as a bridge between Hypnos and Nyx. The companion is able to send messages to and receive messages from Hypnos using Messaging or File-transfer API and communicate with Nyx by establishing a WebSocket connection. Messages related to a session, e.g., starting and ending a

session, should be directly relayed and handled by Nyx as they do not concern the companion. When a binary file containing sensor data is received from Hypnos, it should be transformed to JSON and relayed to Nyx.

5.8.1 Establishing A WebSocket Connection

A WebSocket connection can be established using the WebSocket API available for companion applications. Whenever Nyx is running, a WebSocket server is listening for new connections. In order to connect to this server, we may instantiate a WebSocket object that handles the connection. Once a disconnection occurs, a WebSocket object does not attempt to reconnect. Since our solution should be able to collect sensor data for longer time periods and it is likely that the connection may be lost at some point, we must deal with any disconnection that may occur. In other words, a new connection must be initiated by creating a new WebSocket object with the same configuration. If the connection is lost or any of the participants purposefully close the socket connection, a handler method is called with a parameter that contains a code identifying the reason for the disconnection. In Table 5.4, the closure codes along with explanations are listed. Since only closure code 1000 corresponds to a successful closure acknowledged by both the client and the server, all other codes should be regarded as erroneous. Therefore, for all closure codes except for 1000, reconnection should be attempted.

5.8.2 Companion Lifecycle

Launching Hypnos on the Ionic smartwatch will notify the FMA, which in response should launch our companion application. Consequently, closing Hypnos on the smartwatch should close the companion in the FMA. The companion should, in theory, stay open as long as smartwatch application is running [52]. However, since the FMA is delegated by Android, the companion application may be closed if the host device is running low on resources. To avoid such suspensions, we must make sure that no other process in running on the Android device and make sure that the companion is relaunched at certain intervals.

5.9 Android Application Design

In order to store the sensor readings from Hypnos, we design an Android application called Nyx. The application should be able to collect data from Hypnos, store it persistently in a database, and present it in a structured manner. A

[9] `https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent`

Table 5.4: WebSocket close event codes [9]

Close Event Code	Explanation
1000	Normal close without any errors.
1001	Either client or server left the connection.
1002	Closure due to protocol error.
1003	Closure due to unsupported message data format.
1004/1005	Reserved by the protocol.
1006	Indicates erroneous closure without close frame.
1007	Closure due to encoding mismatch.
1008	Closure due to policy violation.
1009	Closure due to abnormal message size.
1010	Closure due to extension negotiation error.
1011	Closure due to internal error.
1012	Closure due to restart.
1013	Closure due to overload.
1014	Closure due to bad gateway.
1015	Closure due to erroneous TLS handshake.

WebSocket server should run within Nyx to receive sensor readings from the companion. Processing or analyzing the sensor data is not a concern of Nyx.

5.9.1 WebSocket Server

Launching Nyx on the Android smartphone should immediately start the WebSocket server such that Hypnos is able to establish a connection. The server is responsible for handling new connections, processing messages, and performing appropriate action based on the commands received from Hypnos or the companion. In our earlier prototypes, the server was running on the main thread of the application. The most significant issue with this approach is that Android applications will be subject to strict battery-saving policies if they have been running in the background for a certain time period. Applications that consume battery in the background are therefore killed in order to conserve battery. During our experimentation, we received notifications from the system stating that Nyx was closed due to battery usage in the background. When the server is closed, the companion is no longer able to forward sensor readings from Hypnos.

WebSocket Service

In Android, a `Service` is able to run in the background indefinitely if started as a foreground service (Section 4.2.2). Initializing and running the WebSocket server within a foreground `Service` enables us to keep the server running in the background regardless of the application being visible on the screen or the device being awake. Therefore, we do not need to force the device to stay awake since our foreground service will run until it is manually stopped. Consequently, the WebSocket server may run indefinitely throughout longer sensor data collection sessions. Figure 5.9 shows the updated architecture of Nyx where the WebSocket server is running within a service.

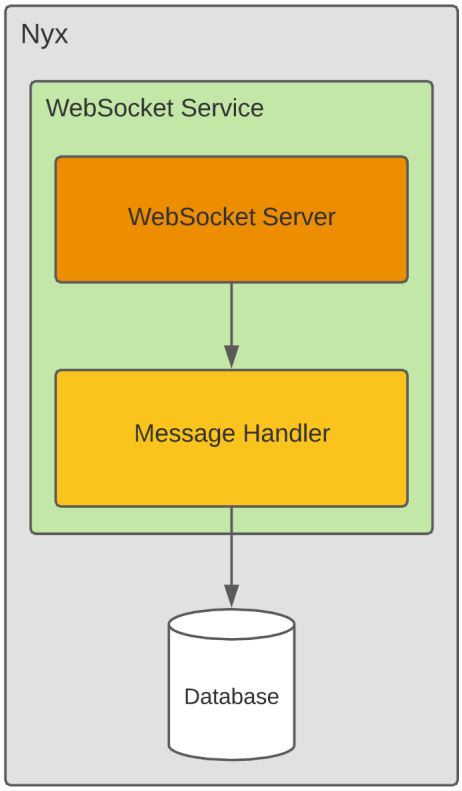


Figure 5.9: WebSocket Service in Nyx.

5.9.2 Data Persistency

Earlier in this chapter, we discussed the limitations of the Ionic smartwatch with regard to internal storage. We concluded that the sensor data must be transferred to and stored in an Android application instead. For this purpose, we must implement a persistency mechanism that is able to meet the requirements of high-volume sensor data. Specifically, the data store must be able to write data rapidly during long-running data collection sessions and not act as a bottleneck for the system. In addition, the data store must allow us to read the data rapidly such that we can export it from the Android application to a computer.

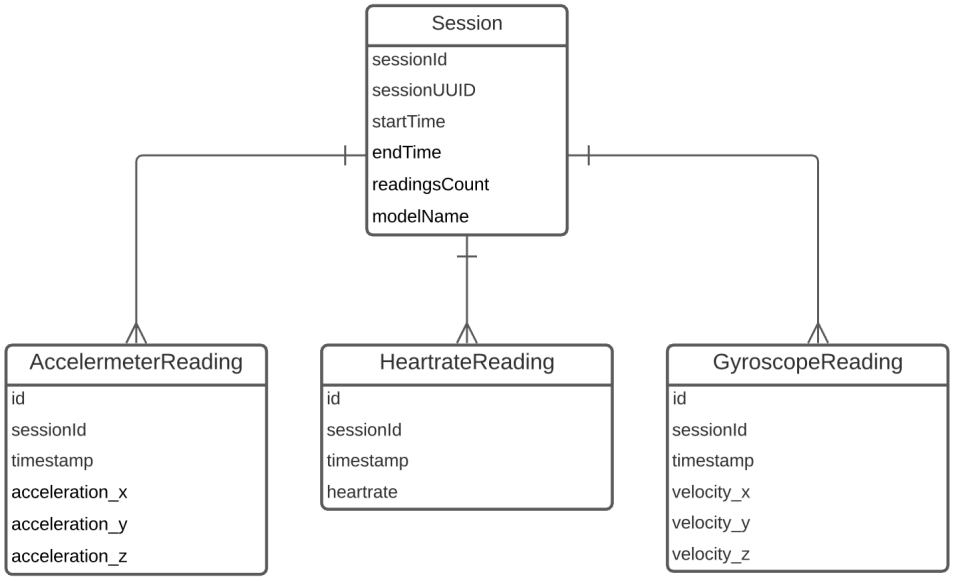


Figure 5.10: Initial database schema for Nyx

Database Modelling

The initial data model consisted of a `Session` entity for storing session data and a variable number of entities for the data from each sensor. Since each sensor outputs data in a different format, data was stored differently across each sensor entity. For example, an accelerometer sensor outputs values for acceleration in three axes, whereas a heart rate sensor only outputs a single value. Therefore, the number of columns varies across each sensor reading entity. Since the readings belong to at most one session, a one-to-many relationship between the session and reading entities is added. In Figure 5.10, we illustrate this schema using

an entity-relationship (ER) diagram. The `Session` entity has an attribute for a unique identifier of the session, attributes that store the starting time and ending time of a session, and an attribute which contains the total number of sensor reading recorded. Further, an attribute stores the model of the Fitbit smartwatch. The entities for the sensors all share common fields such as the timestamp at which the reading was recorded and an identifier for the sensor that performed the reading. A generic class for the readings was created which filtered out the common attributes in a single class.

In order to store data from each sensor, our initial data model had a separate entity for each type of sensor data. This approach is acceptable if the number of sensors is fixed as the data model does not need to change. However, one of our main aims for Hypnos is that new sensors can be added in the future without substantial changes to the source code. To achieve this, we need to take a more generic approach that can cope with additional sensors being added to Hypnos, without requiring a schema redesign. This is important so that, when the built-in oximeter sensor is available, changes should only be made in Hypnos. An improved solution would therefore be to replace the entities for each sensor with a single entity `Reading`, that contains readings from all of the sensors. Since the sensors have a variable number of attributes, merging them into one entity is not possible.

Another issue with this approach is that the configuration of the sensors is not stored. The number of sensors in a session and their frequencies might vary depending on the settings chosen in Hypnos. Nyx should be able to store information about which sensor was enabled during a session. To cope with this, we introduce new entities, Sensor and SessionSensor, that are meant to store which keep track of the sensors and their usage in sessions. Since a session can have multiple sensors and vice versa, the SessionSensor entity acts as a join table having primary keys of both sensors and sessions. In Figure 5.11, we presented the new schema design that copes with a variable number of sensors and a generic entity for storing readings.

Database Performance in Android

During the design of the Android application, we experimented with both relational and non-relational databases for persisting the sensor data. A typical use case for SQLite in Android is caching data such that it can be viewed when the device is offline. If the application displays the data in a list, the data can be polled regularly to only show a subset of the total dataset using paging libraries. For such cases, an SQLite database performs well independent of the size of the database. However, our application must be able to fetch all sensor readings from

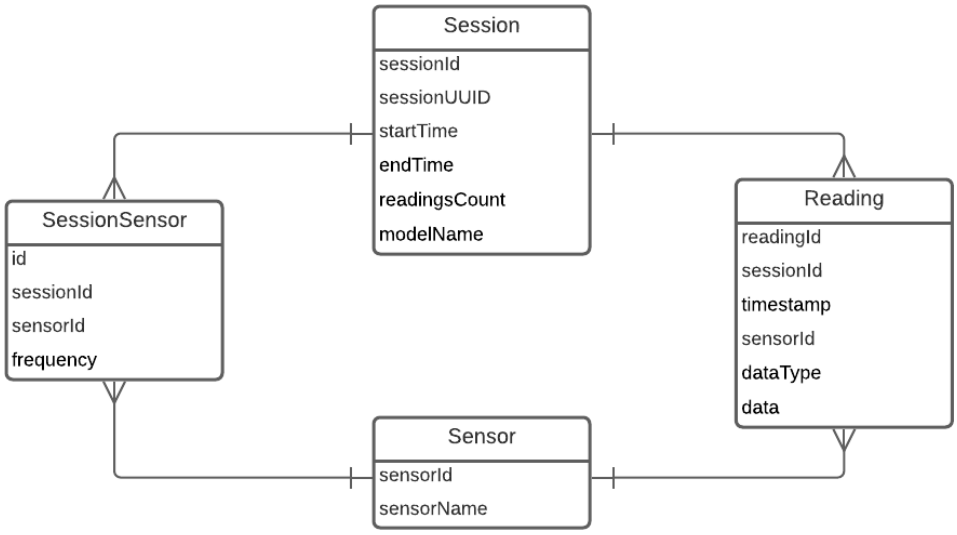


Figure 5.11: Improved schema design.

a session in order to export the data. The number of readings for sessions lasting 8 hours may easily reach beyond millions of entries. The writing performance is not affected by the large amount of data since they may be performed in smaller batches, but reading all collected data points throughout a longer session reaches the limit of the SQLite implementation in Android. On the other hand, a Realm database performs significantly better in terms of read-performance during our preliminary testing. As mentioned in Section 4.4.2, non-relational databases such as Realm store objects directly in memory, instead of mapping objects to and from relations. Our experience with NoSQL as opposed to a relational database in Android is similar to the literature which we presented in Section 4.4.3. Using Realm to store the sensor data yields high performance for both write and read operations and is therefore the database used in our Android application.

Chapter 6

Implementation

In this chapter, we go through the implementation of the applications developed in this thesis. First, we present in Section 6.1 the essential parts of the implementation of our smartwatch application. Then, we present in Section 6.2 the implementation details of the companion application that is bridging Hypnos and Nyx. Finally, we present in Section 6.3 the implementation of the Android application, Nyx.

6.1 Implementation of Hypnos

The smartwatch application, Hypnos, is mainly responsible for collecting readings from the sensors and temporarily storing the data before it is offloaded to the companion application. Starting and ending sessions must be performed from Hypnos, which automatically initiates a new session in Nyx. Further, enabling, disabling, and configuring the frequency of the sensors is also available through Hypnos. In this section, we go through the implementation of the different components of Hypnos.

6.1.1 Initialization

Hypnos consists of multiple views that represent different aspects of the application. All views are defined in separate classes that each extend the `View` class provided by the `ionic-views` framework. Each view class contains methods and instance variables relevant to the screen that is currently visible on the screen. For instance, the `RecordView` is responsible for initiating a new session by notifying Nyx, starting all sensors, and dispatch the sensor data. The `Application` class defines the `Views` of the application as well as properties that are passed to all `Views`. As stated earlier in Section 3.3, a file named `index.js` must be created at the root of the project for the Fitbit application to start. This file contains the code that is executed first and therefore acts as an entry point. We define in this file our entry class `HypnosApplication` by extending the `Application` class and initialize the views. Listing 6.1 shows the `HypnosApplication` class with the `screens` property containing all the `Views` of Hypnos. All `Views` must be

imported as shown in the first few lines of Listing 6.1, before they can be added to the `screens` variable. In this class, we also define `props` property which holds values that should be available for all `Views` of the application. In this property, we store the connection state to Nyx, i.e., if Hypnos is connected to Nyx the `connected` boolean has the value `true` and `false` if Nyx is absent. Since a new session cannot be started if Nyx is unavailable, we use this property to disable the button which starts a session.

Listing 6.1: HypnosApplication class that defines all views of Hypnos.

```
1  import { Main } from './views/main';
2  import { RecordView } from './views/session/recordView';
3  ...
4  import { SamplingTumbler } from './views/settings/sampling/samplingTumbler';
5
6  class HypnosApplication extends Application {
7
8      screens = {
9          Main,
10         RecordView,
11         ...
12         SamplingTumbler
13     }
14
15     props = {
16         connected: false
17     }
18
19     init() {
20         ...
21     }
22
23 }
```

The `init` method is executed once at startup and is used for initializing the application. The contents of the `init` method is shown in Listing 6.2. Since all Fitbit applications are closed automatically after 2 minutes if the user has not interacted with the application, we disable the application timeout in the `init` method. By default, this property is set to `true`, meaning that the application will be closed shortly after launch. Since Hypnos must be open to record sensor data, we set this property to `false` within the `init` method. The sensor configurations, e.g., frequency and state, are stored in a file on disk. In the initialization, we therefore execute the `createPreferencesIfNotExists` method of the `PreferencesManager` such that the sensor configuration file is created if

it does not already exist in the file system. The last two elements in the `init` method send the `START_SEARCH` command handle the connection state received from the companion, respectively.

Listing 6.2: Initialization of application properties and callbacks.

```
init() {
    me.appTimeoutEnabled = false;

    PreferencesManager.createPreferencesIfNotExists();

    messaging.peerSocket.addEventListener("open", (evt) => {
        if (messaging.peerSocket.readyState === messaging.peerSocket.OPEN) {
            let data = cbor.encode({
                command: "START_SEARCH",
                data: {
                    modelName: device.modelName
                }
            });
            messaging.peerSocket.send(data);
        }
    });

    messaging.peerSocket.addEventListener("message", (evt) => {
        switch (evt.data) {
            case "CONNECT":
                this.props.connected = true;
                break;
            case "DISCONNECT":
                this.props.connected = false;
                break;
            default:
                break;
        }
        this.onPropChange(this.props);
    });
}
```

Application Views

Hypnos consists of the `Views` listed in Table 6.1. `Main` is the initially loaded `View` and therefore visible on screen after Hypnos is launched. From this `View`, it is possible to navigate to the `Settings` View in order to adjust sensor configuration or to the `RecordView` to start a new data acquisition session. When a session is finished, the `Summary` screen is shown with the number of readings collected throughout the session. `ToggleSensor` lists all available sensors with checkboxes for enabling or disabling each sensor. Lastly, the `SensorSampling` and `SamplingTumbler` present a list of the frequencies for the sensors and an

interface for changing the frequency of a sensor, respectively.

Table 6.1: Views of Hypnos.

View	Responsibility
Main	Initial view of Hypnos.
RecordView	View for starting and stopping a new session.
Summary	View summarizing the recent session.
Settings	Settings view listing available preferences.
ToggleSensor	View for enabling and disabling of sensors.
SensorSampling	View listing the sensors and frequencies.
SamplingTumbler	Scrollable view for selecting a frequency.

6.1.2 Sensor Implementation

The Fitbit SDK allows all enabled sensors to be imported by including an `import` statement for each sensor. As listed in Table 5.1, the Ionic smartwatch has four sensors that are available and enabled for third-party developers. To use a new sensor in Hypnos, we create a new JavaScript file and import the sensor within this file. Listing 6.3 shows the file created for the accelerometer sensor. This file should contain a function that returns an instantiation of the sensor, along with an identifier and the properties of the readings produced by the sensor.

Listing 6.3: Defining the accelerometer sensor

```
1 import { Accelerometer } from "accelerometer";
2
3 export const accelerometer = () => {
4     return {
5         sensor: new Accelerometer(),
6         identifier: "ACCELEROMETER",
7         properties: ["x","y","z"]
8     }
9 }
```

Then, the sensor file should be imported within a file called `sensorObjects.js` and appended to the `SENSORS` constant within this file along with all other sensors. Sensors that are defined here are able to be initialized, configured, and controlled by the `SensorManager`. Listing 6.4, shows the various sensors that are imported.

Listing 6.4: The sensorObjects.js file in Hypnos

```
import { accelerometer } from './sensors/accelerometer';
import { barometer } from './sensors/barometer';
import { battery } from './sensors/battery';
import { gyroscope } from './sensors/gyroscope';
import { heartrate } from './sensors/heartrate';
import { memory } from './sensors/memory';

export const SENSORS = [
    accelerometer,
    gyroscope,
    heartrate,
    battery,
    memory,
    barometer
];
```

Finally, each sensor should be defined in the `sensorDefinitions.js` as shown in Listing 6.5. This file is used to create a preference file in which the sensor states and frequencies are stored. It is important that the `sensor` property is identical to the identifier defined in Listing 6.3. By doing so, we should be able to configure the sensor instance according to the preferences set in Hypnos. The `displayName` property defines how the sensor is shown within the Hypnos preferences. The last two properties, `typeCount` and `types`, define the number of properties a sensor has and the labels that should be used for the properties within Nyx, respectively.

Listing 6.5: The sensorDefinitions.js file

```
export const SENSOR_DEFINITIONS = [
    {
        "sensor": "ACCELEROMETER",
        "displayName": "Accelerometer",
        "typeCount": 3,
        "types": [
            "X",
            "Y",
            "Z"
        ]
    },
    ...
]
```

6.1.3 SensorManager

The `SensorManager` is responsible for controlling the sensors. It contains methods for starting and stopping all sensors at once, enabling and disabling the sensors, and setting the correct frequency based on user settings. Internally, it uses

the `SENSORS` constant defined in the `sensorObjects.js` file, shown in Listing 6.4, and the applications preferences to create an array of the sensors which should be active. When a session is initiated, only the sensors in this array are started. In Listing 6.6, we show the configuration of sensors in the `SensorManager`. First, we define the sample rate by using the `setOptions` method, then we define the callback method which should be executed each time the sensor delivers new readings. Notice that a sensor is only configured and added to the `enabledSensors` array if the `enabled` property is `true`.

The configuration of the sample rate requires an object with two keys: `frequency`, which specifies the sampling rate of the sensor, and `batch`, which specifies the number of readings delivered at once. The frequency is set according to the preferences used in the application settings and is 1 by default. If the `batch` key is omitted, each new reading will have to be handled individually. For lower frequencies, e.g., 1 to 5 Hz, handling each data individually should not cause any issues. However, since some sensors may be configured with higher frequencies, e.g., 20 to 40 Hz, handling each data individually will block the event queue due to a large number of readings. To avoid this, we may set the `batch` to be a multiple of the `frequency` value. For example, if the multiple is 1, readings are delivered each second regardless of the frequency of the sensor, and if the multiple is 2, readings are delivered every other second. Due to memory concerns, we use a multiplier of 2 such that the number of readings handled in the memory is no more than twice the frequency.

Listing 6.6: Sensor configuration in the SensorManager

```
1  constructor(handler) {
2      SENSORS.forEach((s) => {
3          const {sensor, identifier, properties} = s();
4          let enabled = this.prefManager.getSensorStatus(identifier);
5
6          if (enabled) {
7              const f = this.prefManager.getSensorFrequencyFor(identifier);
8              sensor.setOptions({
9                  frequency: f.frequency,
10                 batch: f.frequency < 1 ? 1 : Math.round(f.frequency * 2)
11             });
12
13             sensor.onreading = () => {
14                 let data = [sensor.readings.timestamp];
15
16                 properties.forEach((prop) => {
17                     data.push(sensor.readings[prop])
18                 });
19                 handler(identifier, data);
20             }
21             this.enabledSensors.push(sensor);
```

```
22            }
23        });
24    }
```

6.1.4 Caching to Disk

In Section 5.5, we discussed multiple architectures for caching sensor data to lower the transmission rate. Ultimately, caching sensor data to disk in binary files is more favorable as the Ionic smartwatch has a limited amount of memory. Although binary files are more complicated to construct, it is the most efficient format in terms of both storage space occupied on disk and transfer speed, since there are no data descriptors that increase the size of the file.

Binary File Structure

A binary file may be created from an `ArrayBuffer` by allocating the fixed number of bytes. The bytes allocated should be calculated using the dimensions of the data we wish to store, i.e., the number of columns for the sensor reading properties and the rows that correspond to the number of readings. Since all elements of the buffer must contain exactly the same amount of bytes, the calculation can be performed by multiplying the columns and rows with a fixed number of bytes. In order to insert data into the `ArrayBuffer`, we must use a typed array where the type should be able to store the sensor data. According to the sensor documentation [53], all hardware sensors deliver data as 32-bit floating-point numbers when readings are delivered in a batch. 32-bit floating-point numbers may be stored in a Float32Array which stores elements in 4 bytes.

However, since we also store a timestamp along with each reading, and a timestamp in milliseconds requires more than 4 bytes, the Float32Array will not be able to store the timestamps correctly. Furthermore, since we cannot vary the byte size depending on the elements, we either have to store all values in 8 bytes or reduce the bytes needed to store the timestamp. To achieve the latter, one approach would be to use the initial session timestamp as a reference to calculate the time delta, i.e., the first reading has a timestamp equal to 0 and the last reading has a timestamp equal to the duration. The largest possible integer value in a 32-bit-floating-point is 16777216, i.e., the delta can be at most 16777216 milliseconds, which corresponds to approximately 4.6 hours. Therefore, we cannot use Float32Array to store longer sessions. To store higher integers values we must use a Float64Array, which has practically no limit on the timestamps.

Therefore, the total bytes needed to allocate is equal to 8 bytes multiplied by the sensor reading properties and the number of readings. To exemplify,

we may study sensor readings from the accelerometer sensor which has 4 types: timestamp, x acceleration, y acceleration, and z acceleration. In Table 6.2, we have listed 4 accelerometer readings. Since the total elements to be stored are 16 (rows multiplied by columns), we must allocate 128 bytes for the `ArrayBuffer` to store the readings in Table 6.2.

Table 6.2: Table representation of accelerometer readings.

	timestamp	x	y	z
1	1618489403050	-0.092179298400	8.010046005249	5.664853096008
2	1618489403100	-0.079010963439	8.004059791564	5.682810783386
3	1618489403150	-0.104150772094	8.005256652832	5.692387580871
4	1618489404250	-0.104150772094	8.031594276428	5.678022384643

In order to write the readings to disk as binary files, we must iterate the readings and the data types to insert the data in the corresponding location of the `ArrayBuffer`. To do this, we calculate the correct index by using the row number and the offset from start. This calculation can be done as shown inside the nested for loop in Listing 6.7. The variable `i` is the current reading and `count` is the number of reading properties.

Listing 6.7: Inserting reading data into the ArrayBuffer.

```
1 for (let i = 0; i < n; i++) {
2     for (let k = 0; k < count; k++) {
3         let b = count * i + k;
4         bytes[b] = readings[k][i];
5     }
6 }
```

Writing Binary Files

After the bytes are added, the `ArrayBuffer` can be written to disk using the `writeSync` method of the FS API. This method takes the file reference and the buffer as first and second parameters, respectively. A file reference can be obtained by opening the file as shown in line 2 of Listing 6.8. Passing `a+` as a parameter means that we may append to an already existing file and accumulate fewer files to send to the companion. After readings have been added, we call the `closeSync` method to close the open file.

Listing 6.8: Method for appending sensor data to disk.

```
1 appendToFile(fileName, buffer) {
2     let file = fs.openSync(fileName, "a+");
3     fs.writeSync(file, buffer);
```

```
4       fs.closeSync(file);
5   }
```

6.1.5 Handling Sensor Readings

Although the sensor readings vary in the number of properties, they all have (1) a property for the timestamp of the readings (2) and additional properties. Therefore, we may create a generic method for handling the data of the sensors as can be seen in Listing 6.9. This method takes two parameters: a value containing the identifier of the sensor, e.g., ACCELEROMETER, and a multidimensional array of the readings. The multidimensional array must have a timestamp array as the first element and an arbitrary number of elements for the sensor properties. Using the elements of these arrays, we calculate the size of the `ArrayBuffer`. This calculation is performed on line 6 in Listing 6.9. The readings are then inserted in the `ArrayBuffer` by looping over the readings and the properties of each reading. The `ArrayBuffer` is then written to a file on disk. Since the content of the file is binary, the name of the file is used to specify the sensor type instead. We use the `sensor` parameter along with a timestamp to give the file a distinct name. A counter is used in order to batch multiple `ArrayBuffer`s into a single file, reducing the number of files needed to be dispatched. When the counter reaches the `batchLimit` property, it is added to the `DispatchManager` and a new file is created.

Listing 6.9: Generic method for persisting sensor data to disk.

```
1   readingHandler(sensor, values) {
2       let now = Date.now();
3       let n = (values[0]).length;
4       let count = values.length;
5
6       let buffer = new ArrayBuffer(8 * count * n);
7       let bytes = new Float64Array(buffer);
8       values[0] = this.adjustTimestamp(values[0], now);
9
10      for (let i = 0; i < n; i++) {
11          for (let k = 0; k < count; k++) {
12              let b = count * i + k;
13              bytes[b] = values[k][i];
14          }
15      }
16
17      let file = this.sensorFiles[sensor];
18
19      if (file.counter > this.batchLimit) {
20          this.dispatchManager.addToQueue(file.fileName);
21          file.fileName = `${sensor}.${Date.now()}`
```

```
22              file.counter = 0;
23          }
24          this.appendToFile(file.fileName, buffer);
25          file.counter += 1;
26
27          this.eventCount += n;
28      }
```

The hardware sensors use a randomized value as a reference for the timestamps of the readings. Therefore, we must convert the timestamps to *epoch time*. Since the readings are sorted by timestamp in ascending order, we may register a *reference* time as soon as the `readingHandler` method is called and calculate the difference between the reference and the sensor time stamps. For example, the last reading in a batch should use the reference timestamp registered in the second line of Listing 6.9. Then, the second to last reading should subtract the difference to the first reading from the reference timestamp and so on. In Listing 6.10, the `adjustTimestamp` method that performs the conversion from the arbitrary timestamps to epoch timestamps is shown. The timestamps of the readings and the reference timestamp must be passed as parameters. This approach assumes that the `readingHandler` method is called right after the batch is full so that the last timestamp is identical to the reference. If there is any delay between the arrival of the last reading and the call to `readingHandler`, this may affect the accuracy.

Listing 6.10: Method used to convert the reading timestamps to epoch.

```
1   adjustTimestamp(timestamps, reference) {
2       let ts = [];
3       let i = timestamps.length − 1;
4       let j = i;
5       let lastTs = timestamps[i];
6       do {
7           ts.push(reference − lastTs + timestamps[j−i])
8       } while(i−−);
9       return ts;
10  }
```

6.1.6 Virtual Sensors

In addition to the physical sensors, e.g., accelerometer, gyroscope, altimeter, we may create virtual sensors. By utilizing additional APIs, we may create a sensor that can collect data similar to the sensors provided by the SDK. For instance, the SDK offers a Battery API for retrieving the current battery percentage as shown in Listing 6.11. In order to create a virtual sensor, we define a generic

sensor class that can be extended to create additional sensors. The generic class handles the frequency, batch size, and reading handlers similar to the hardware sensors. A new virtual sensor class must extend the `GenericSensor` class and override the `getReading` method.

Listing 6.11: Retrieving battery percentage from the Power API

```
import { battery } from "power";
let percentage = battery.chargeLevel;

console.log(percentage); // Prints current battery level to the console
```

Listing 6.12 shows the implementation of a battery sensor which is created using the `GenericSensor` class. This sensor can be imported and utilized identical to the accelerometer sensor shown in Section 6.1.2. A battery sensor may be used to log the battery level throughout a long session to make sure that Hypnos does not draw unusual amounts of battery. It also demonstrates that the `GenericSensor` class we implemented, can be used to easily create new virtual sensors. Furthermore, the SDK includes an API for retrieving the memory usage which can, similar to the battery sensor, be used to create a memory sensor. A memory sensor is important as it can be used to log the memory usage throughout a session and ensure that we do experience a memory leak at any point in the session.

Listing 6.12: Battery sensor implementation.

```
import { battery } from "power";
import { GenericSensor } from "./GenericSensor";

export class Battery extends GenericSensor {

    getReading() {
        return battery.chargeLevel;
    }

}
```

6.1.7 Dispatch Manager

The acquired sensor data is written to disk continuously throughout a session but dispatched only at certain intervals using the `DispatchManager`. This class is responsible for keeping an array of files that are to be sent to the companion. Internally, the `DispatchManager` uses the File-transfer API to send the sensor reading files. In Listing 6.13, the configurable instance variables are shown. The `files` array keeps track of the files that are to be sent in the next dispatch.

The `filesLimit` variable determines the number maximum of sensor data files that should be buffered. The `delay` and `firstDelay` are used to determine the delay between each file dispatch and the initial delay before the first file is dispatch, respectively. During the development of the `DispatchManager`, we experimented with different values for these variables. Setting `delay` too low blocks the event loop such that other events, i.e., new readings from sensors, are delayed. Our experimentation resulted in a 500 milliseconds delay which is long enough for the smartwatch to process new sensor readings between the dispatches. The `dispatchRate` is used to determine the delay between each dispatch round. Every `dispatchRate` milliseconds, the files containing the sensor readings in the array `files` are sent to the companion. Similar to `delay`, the `dispatchRate` variable must be set considering the limitations of the File-transfer API. In our implementation, we set this value to 60 seconds.

Listing 6.13: Instance variables of the Dispatch Manager

```
1  export default class DispatchManager {
2
3      files = [];
4      filesLimit = 40;
5      delay = 500;
6      firstDelay = 30;
7      dispatchRate = 60000;
8      dispatchInterval;
9
10     ...
11 }
```

The dispatch of sensor data from disk can be initiated by called the `start` method of the `DispatchManager`, as shown in Listing 6.14. When a new session is started, the `start` method of the `DispatchManager` is called. This method starts an interval that sends data every `dispatchRate` milliseconds by calling the `dispatch` method. The interval is a conventional JavaScript interval that takes one parameter for a callback function that should be invoked and another for the delay between each call.

Listing 6.14: Method used for starting the DispatchManager.

```
1  start() {
2          this.dispatchInterval = setInterval(() => {
3              this.dispatch();
4          }, this.dispatchRate);
5  }
```

The interval is referenced through the `dispatchInterval` variable such that it can be canceled when the session is ended. The `stop` method of Dispatch-

Manager, shown in Listing 6.15, uses the `clearInterval` method to cancel the interval, which essentially stops the continuous dispatch of sensor data to the companion. Both `setInterval` and `clearInterval` are standard JavaScript methods and not implemented in the `DispatchManager`.

Listing 6.15: Method used for stopping the DispatchManager.

```
stop() {
        clearInterval(this.dispatchInterval);
}
```

The `dispatch` method, which is shown in Listing 6.16, uses the File-transfer API to send data to the companion application. Each call to `dispatch` creates a copy of the `files` array with the sensor file names and then clears it. The copy of the `files` array is iterated and added to the `outbox` queue of the File-transfer API that handles the dispatch. If the file is successfully sent and received on the companion, it is deleted from the smartwatch using the `unlinkSync` method. On the other hand, if the file transmission fails, the file is stored on the smartwatch until the next round of dispatch. Using the `enumerate` method, we check to see if there are any failed transfer attempts and cancel them if there are any. Then, if the file exists on disk, it should be pushed the `files` array and attempted to send again at the next call to `dispatch`.

Listing 6.16: Method for sending data to the companion.

```
dispatch() {
    let dispatchQueue = this.files.slice(0, this.files.length);
    this.files = this.files.splice(this.files.length);

    console.log(`Dispatching ${dispatchQueue.length} items...`)

    outbox.enumerate().then((files) => {
        console.log(`Files to be sent: ${files.length}`)

        files.map((file) => {
            // Remove corrupt file transfer
            file.cancel();

            // If available in the file system, then reschedule send
            if (existsSync(`/private/data/${file.name}`)) {
                this.addToQueue(file.name);
            }
        })
    })

    for (let i = 0; i < dispatchQueue.length; i++) {

        setTimeout(() => {
                let path = `/private/data/${dispatchQueue[i]}`;
```

```
25
26                  outbox.enqueueFile(path).then((ft) => {
27                      // Successfully transferred files should be deleted
28                      // to free space for new data.
29                      unlinkSync(path);
30
31                  }).catch((error) => {
32                      console.log(‘Failed to schedule transfer: ${error}‘);
33                  })
34
35          }, this.delay*i + this.firstDelay);
36      }
37  }
```

6.2 Implementation of Companion Application

The companion application runs on the connected Android smartphone simultaneously as Hypnos. The purpose of the companion is to enable communication between the two devices such that data transfer can occur. In our implementation, we use the companion to send commands from Hypnos to Nyx regarding the session that is to be recorded as well as files that contain sensor data.

6.2.1 Bridging Hypnos and Nyx

Since the Bluetooth connection between the smartwatch and smartphone is restricted to the File-transfer and Messaging APIs, we must use the companion as a bridge to relay messages and data. Sending data from the smartwatch to the companion is done by using the aforementioned APIs. Propagating them to the smartphone requires that we establish a WebSocket connection, between the companion and Nyx. As the companion (client) and Nyx (server) are running on the same device, we may access the server by connecting to the loopback IP of 127.0.0.1. Normally, we should connect to a WebSocket server using an encrypted connection, but since the server is local, we use an unencrypted connection.

WebSocketHandler

A WebSocketHandler class is implemented to establish a new WebSocket connection. This class attempts to reconnect to the server every `_reConnectDelay` seconds until the server is available, i.e., until Nyx is launched on the Android device. Using only the WebSocket class provided by the Companion API, we are not able to reconnect if a disconnection occurs or if Nyx is launched after Hypnos. This is because the standard WebSocket does not attempt a reconnec-

tion if the connection fails. In Listing 6.17, we create a new WebSocketHandler that should connect to the Nyx server using the loopback address and a port of 8887. The port may be defined arbitrarily but should be identical to the port used by the server. A connection is only attempted when the `start` method of the `WebSocketHandler` is explicitly called.

Listing 6.17: Instantiation of a WebSocketHandler.

```
1 import { WebSocketHandler } from "./WebSocketHandler";
2
3 const port = 8887;
4 const host = "127.0.0.1";
5
6 var websocket = new WebSocketHandler(host, port);
```

The `WebSocketHandler` uses the host address and port number to create a Uniform Resource Identifier, or URI, of the following format: `ws://host:port/`. If the WebSocket server is remote, the host may be substituted with the IP address of the remote server. Since the companion app and the server are local, we use the loopback address. The `start` method, which is called after the reception of the `START_SEARCH` command, initiates the connection to the server in Nyx. Inside `start`, a new connection is attempted by creating a WebSocket object. Figure 6.1 shows the connection procedure between Hypnos and Nyx as a sequence diagram. As can be seen from this figure, the WebSockerHandler is instantiated immediately since Hypnos and the companion are launched simultaneously, but a WebSocket object is only created after `start` is called. Depending on the result of the connection attempt, either the open handler is called on both ends or a new connection is attempted by calling `reConnect`. The `reConnect` method will invoke `start` after `_reConnectDelay` milliseconds. Listing 6.18 shows the `reConnect` method where a reference to a JavaScript time out is stored in the `_timeOut` variable such that it can be canceled if the connection is successful. If the server is still unavailable, the reconnection is attempted after the `_reConnectDelay` milliseconds.

Listing 6.18: Method for attempting a reconnection.

```
1 reConnect() {
2     this._timeOut = setTimeout(() => {
3         console.log(`Reconnecting in ${this._reConnectDelay} ms...`)
4         this.start(true);
5     }, this._reConnectDelay);
6 }
```

In Figure 6.2 (a), Hypnos is waiting for the companion to connect to the WebSocket server in Nyx. When the connection is established, new sessions

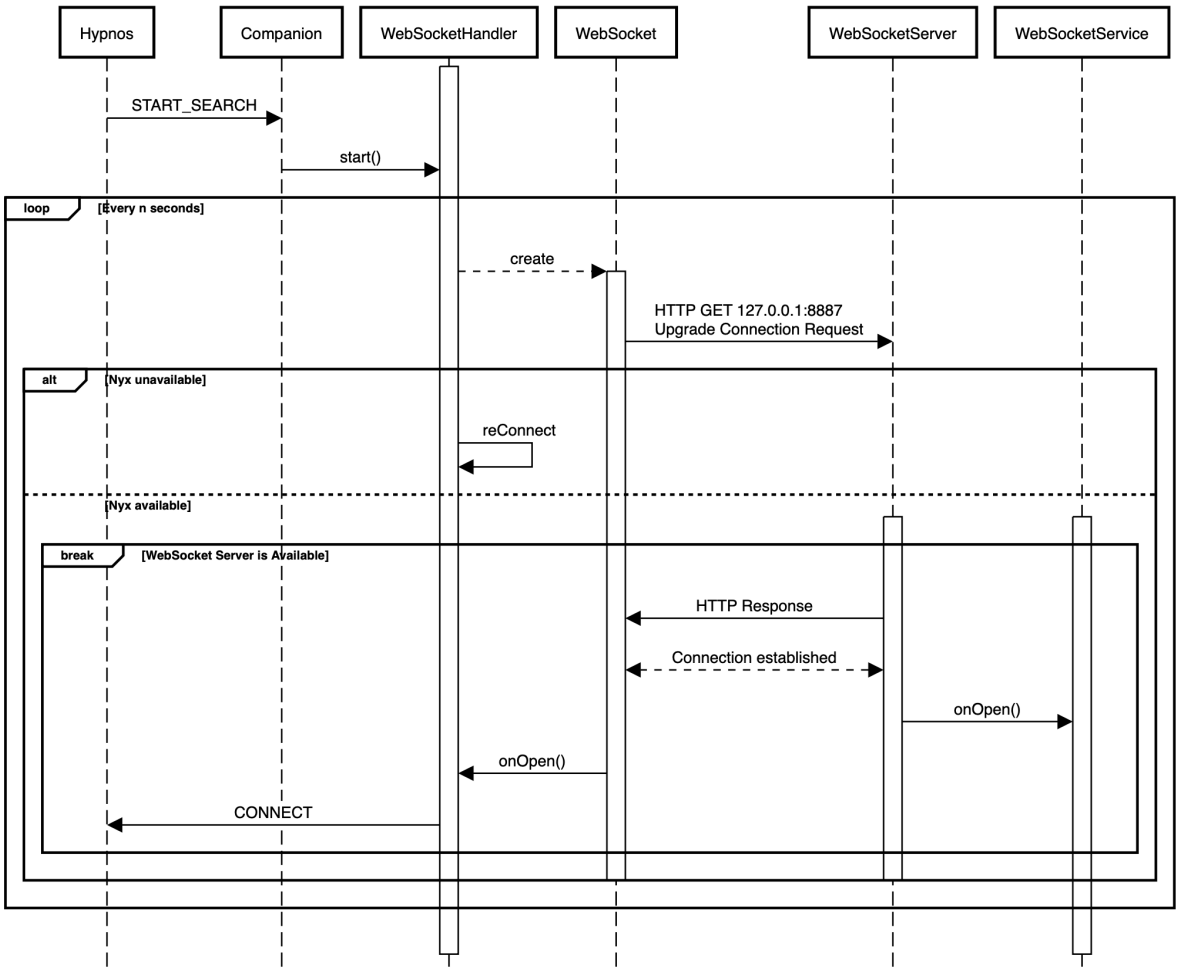
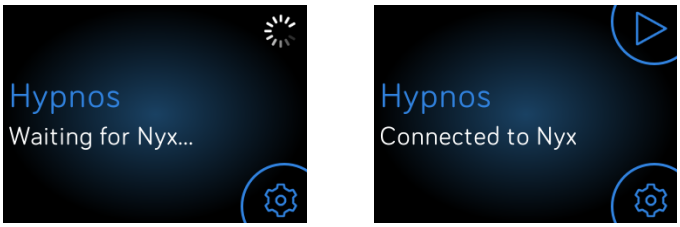


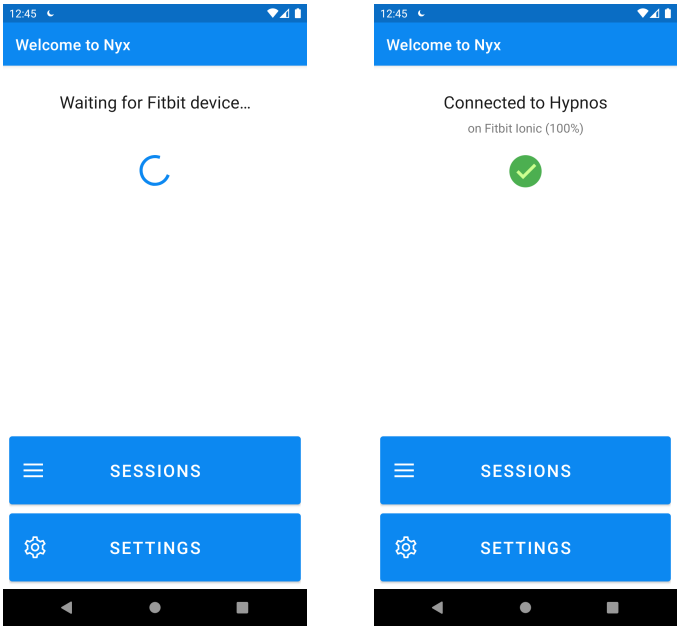
Figure 6.1: Connection procedure between Hypnos and Nyx.

may be started by pressing the button in the upper left corner of Figure 6.2 (b). Similarly, Figure 6.3 shows Nyx both when Hypnos is unavailable, (a), and available, (b).



(a) Nyx is unavailable (b) Nyx is available

Figure 6.2: Hypnos running on the Fitbit Ionic



(a) Hypnos is unavailable (b) Hypnos is available

Figure 6.3: Nyx running on the Android device

6.2.2 Handing Commands

When the companion has connected to both Hypnos and Nyx, it should relay commands received from Hypnos to Nyx. For some commands, the companion

should also perform specific actions. For this, we use an event listener that performs the correction actions based on the message received. Listing 6.19, shows the commands being handled by a `switch` statement. As can be seen in the `switch` statement, when the `START_SEARCH` command is received, the companion attempts to connect to the WebSocket server. If the server is available, a message is sent back to Hypnos such that a new session can be initiated. When `INIT_SESSION` is received, the companion stores the UUID of the session and relays the message to Nyx. The session UUID is stored in case the companion is suspended by Android, which may occur if the host device is running low on resources as stated earlier in Section 5.8.2. The commands `START_SESSION` and `STOP_SESSION` are received when a session is started and stopped, respectively.

Listing 6.19: Message handler for commands received from Hypnos

```
1  messaging.peerSocket.addEventListener("message", (evt) => {
2      let message = cbor.decode(evt.data);
3      let command = message.command;
4
5      switch (command) {
6          case "START_SEARCH":
7              websocket.setOnOpen(() => {
8                  messaging.peerSocket.send("CONNECT");
9                  ...
10             })
11             websocket.start();
12             break;
13         case "STOP_SEARCH":
14             websocket.stop();
15             break;
16         case "INIT_SESSION":
17             uuid = message.payload.sessionIdentifier;
18             localStorage.setItem("sessionUUID", message.payload.sessionIdentifier);
19
20             websocket.send(JSON.stringify(message));
21             break;
22         case "START_SESSION":
23             sessionInProgress = true;
24             websocket.send(JSON.stringify(message));
25             break;
26         case "STOP_SESSION":
27             sessionInProgress = false;
28             websocket.send(JSON.stringify(message));
29             break;
30         default:
31             break;
32     }
33 });
```

6.2.3 Relaying Sensor Data

The File-transfer API sends sensor files asynchronously, i.e., the companion must listen for incoming files continuously and handle them when received. As described in Section 6.1.4, files are stored in a binary format on the smartwatch. Therefore, they only contain the raw readings and no information regarding the sensor or session. Instead, we use a special naming convention that consists of the sensor identifier along with a timestamp. This way, we know what sensor the data belongs to and may therefore read the file accordingly to the dimension of the data. Also, each name will be (practically) unique, since they include a timestamp. For example, readings from the accelerometer sensor may be stored in a file named: `ACCELEROMETER.1618489403050`. Having unique names for each file ensures that we may have multiple files for the same sensor on the smartwatch.

Although the companion is able to handle the binary data, the readings should be transformed to JSON such that Nyx is able to add them to the database. Similar to nested `for` loops writing the sensor data to an `ArrayBuffer` in Listing 6.7, the contents of the `ArrayBuffer` received at the companion is transferred back into a multidimensional array `data`. Then, an `ADD_READING` command is created using the `createReading` function by passing the `data` array as shown in Listing 6.20. The command is stored in the `reading` variable and has a structure identical to that shown in Listing 5.4. Finally, a check is performed if the data should be sent to Nyx or buffered temporarily depending on the connection state.

Listing 6.20: Creation of reading commands using the sensor readings received from Hypnos.

```
1 let reading = createReading(uuid, sensor, types, data);
2
3 if (!isBuffering) {
4     websocket.send(JSON.stringify(reading));
5 } else {
6     let sessionBuffer = JSON.parse(localStorage.getItem("sessionBuffer"));
7     sessionBuffer.buffer.push(reading);
8     localStorage.setItem("sessionBuffer", JSON.stringify(sessionBuffer));
9 }
```

6.3 Implementation of Nyx

The Android application, Nyx, is responsible for persisting data from Hypnos to a database. For this purpose, it must continuously run a WebSocket server and be able to understand the commands received from Hypnos. The implementation of Nyx ensures that the sensor data may be stored indefinitely while at the same time allowing the data to be exported to external locations for further analysis.

6.3.1 WebSocketService

Applications in Android, if left open for a while, will be suspended to conserve battery by halting the CPU. The OS must enforce strict battery-saving policies due to the limited resources of smartphones. For normal usage, this pattern works as intended: when the device is in use or connected to power, no measures are taken, but when the device is put to sleep, applications are suspended after some time to extend battery life. In Android version 6.0 and above, Android uses the *Doze* paradigm which limits execution to time certain intervals [54]. Applications that need to perform background tasks when the device is inactive, must wait for these intervals to for example perform network requests or other activities. Figure 6.4 illustrates the Doze paradigm with green areas represent the idle states and orange spikes represent collective maintenance where applications are allowed to run. However, applications that must run continuously to collect data from external devices, similar to our application Nyx, will not be able to perform such tasks if the application is suspended. Further, since the interval is not consistent, the periods between wakes may be longer than what is possible to cache in Hypnos. During our initial testing of Nyx, we were able to observe disconnections that were a result of this feature.

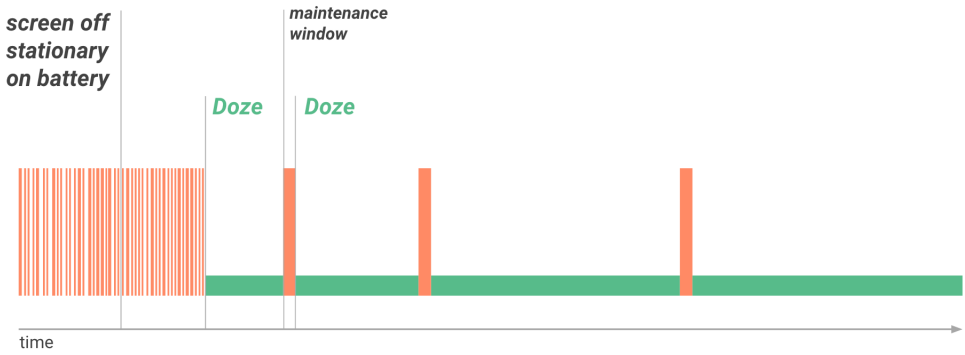


Figure 6.4: Doze paradigm used in Android 6.0 and later [10]

Therefore, we implement a foreground service that is able to stay open despite running on a device with a recent version of Android. Foreground services are prioritized by the system and will continue running until they are manually suspended. To initiate a foreground service, a notification must visible in the top menu bar of the screen at all times to indicate that a background process

[10] `https://developer.android.com/training/monitoring-device-state/doze-standby`

is running. The foreground service is started in the `onCreate` method of the WebSocketService. When the `Service` is terminated, the foreground service is disabled to conserve battery when the application is not in use.

In the first `Activity` of Nyx, we initialize and start a new WebSocketService. If started normally, it will run on the application main thread and possibly cause delays in the UI. We therefore start the `WebSocketService` on a separate thread using an `Executor`, as shown in Listing 6.21. This thread is started subsequently after Nyx is launch and is therefore able to receive sensor data from Hypnos even if the Nyx is running in the background.

Listing 6.21: Starting a thread with a WebSocket Service

```
1  ...
2  private Intent websocketServerServiceIntent;
3
4  @Override
5  protected void onCreate(Bundle savedInstanceState) {
6      super.onCreate(savedInstanceState);
7      setContentView(R.layout.activity_main);
8      ...
9      websocketServerServiceIntent = new Intent(
10             context, CustomWebSocketService.class);
11
12     Executor executor = Executors.newSingleThreadExecutor();
13     executor.execute(new Runnable() {
14         @Override
15         public void run() {
16             startService(websocketServerServiceIntent);
17         }
18     });
19 }
```

6.3.2 WebSocket Server

A WebSocket server is implemented to receive messages and sensor data from Hypnos. Since Java is the main development language for Android, Nyx is developed entirely in Java. To the best of our knowledge, Java does not implement the WebSocket protocol that we need to establish a connection between the companion and Nyx. It is possible to create traditional socket connections using Java, but the Companion API does not support traditional sockets. Therefore, we opted for an open-source implementation of the WebSocket protocol. During the development of Nyx, we investigated the available implementations for Java. From earlier, we had experience with the Java-WebSocket [55] implementation in a previous project. This implementation supports both connecting to a WebSocket server by creating a client and vice versa. A custom server can

be created by extending the `WebSocketServer` class and overriding its methods. The methods that should be overridden are as follows:

- **`onStart`** is called when the server has established a connection to a client. We use this method to update the UI to indicate that the companion is successfully connected to the WebSocket server.
- **`onMessage`** is called when a message is received from the client. The message can be both commands or sensor readings. The messages are handled by a separate class `MessageHandler` which is able to perform the corresponding actions to each command.
- **`onClose`** is called when the connection is closed, either because the client left intentionally or the connection was lost abruptly. In our case, this method will be called if the companion is closed. The server will continue running in case the companion attempts to connect again.
- **`onError`** method indicates that an error has occurred. The parameter contains an `Exception` object that is printed to the logs.

Since the `WebSocketServer` is running within the `WebSocket Service`, these methods are handled within the `Service` using a callback interface as shown in Listing 6.22. The interface is implemented by the `WebSocketService` and the appropriate methods are called from the `WebSocketServer`. A benefit of this approach is that the `Service` may perform actions based on the handlers that are otherwise not possible from a WebSocket server, e.g., handle updates to the UI.

Listing 6.22: WebSocketCallback interface.

```
1 public interface WebSocketCallback {
2     void onOpen();
3     void onClose(int code);
4     void onMessage(String message);
5 }
```

6.3.3 Data Persistency Interface

The database in Nyx should store information about each session, e.g., when it is initiated and ended, what type of device is propagating the sensor data, and the total number of readings. Since the sensor configuration can vary between sessions, we should also store this information in the database. Most importantly, the database should be able to store the readings generated by

the sensors. However, the storage component of Nyx should be configurable to use another database in the future and must therefore be easily modified without changes to the rest of the application. To enable this, we create an interface `NyxSessionStore` which defines the base methods for persisting the data described above to an arbitrary storage solution. Classes that handle the storage of session and sensor data must therefore inherit this interface and implement its method. In Listing 6.23, we present the NyxSessionStore interface and its methods. The interface contains the following abstract methods that must be implemented by a class that is responsible for storing data: `initSession`, `startSession`, `stopSession`, and `addReading`.

- **`initSession`** The `initSession` method is called when Nyx receives a request for a new session from Hypnos. Initiating a new session from Hypnos will propagate information regarding the sensor configurations in Hypnos. Therefore, this method specifies a parameter `sensorConfigs` that contains a key-value pair of the active sensors and the configured frequency. In addition, a UUID for the session and the device model identifier should be passed as parameters.
- **`startSession`** The `startSession` method is called when Hypnos starts collecting data from the sensors. The method has two parameters, sessionIdentifier, which denotes the UUID of the session initialized in `initSession` and startTime, the exact starting time of a session as an epoch timestamp.
- **`stopSession`** When the ongoing session is ended from Hypnos, a command is received in Nyx with the UUID and the timestamp at which the session was stopped. These values are passed as parameters to the `stopSession` method which should correctly persist to a database.
- **`addReading`** Sensor readings received from the companion will invoke the `addReading` method of the `NyxSessionStore` interface. The method has multiple parameters, the first one denoting the session UUID, the second parameter is the sensor that recorded the reading, and lastly, a list of reading objects.

Listing 6.23: NyxSessionStore interface

```
1 public interface NyxSessionStore {
2
3     void initSession(
4     String deviceModel,
5     String sessionIdentifier,
6     HashMap<String, Float> sensorConfigs);
7
```

```
8       void startSession(String sessionIdentifier, long startTime);
9
10      void stopSession(String sessionIdentifier, long endTime,
11      int readingsCount);
12
13      void addReading(String sessionIdentifier, String sensorIdentifier,
14      List<Reading> readings);
15  }
```

6.3.4 Realm Database

The session and sensor data are persisted to a local Realm database. Data may be permanently stored on the smartphone, but since multiple sessions may occupy a lot of the internal storage, it should also be possible to export data from the database. The persisted data must also be accessible from the device itself such that it may be previewed.

Initialization of Realm

After Nyx is launched, the Realm database is initialized within the `NyxApplication` class. Initializing Realm from this class ensures that at most one configuration exists throughout the execution of the application. As can be seen in Listing 6.24, a name is used to reference a Realm instance and the reference is used to build and configure the database at launch. We may, if needed, initialize multiple Realms from this class, but for the purposes of storing sensor data, we only need one.

Listing 6.24: Initialization of Realm.

```
1   public class NyxApplication extends Application {
2
3       @Override
4       public void onCreate() {
5           super.onCreate();
6
7           Realm.init(this);
8           String realmName = ”SessionStore”;
9
10          RealmConfiguration config = new RealmConfiguration
11              .Builder()
12              .name(realmName)
13              .build();
14
15          Realm.setDefaultConfiguration(config);
16      }
17  }
```

Data Models

In order to store sessions, sensors, and sensor readings, model classes are created for each. An important goal when designing the data models is that extending Hypnos with novel sensors should not require any changes to the implementation of Nyx, i.e., the readings received from the sensors must be stored in a generic manner such that any type of data may be persisted. We ensure this by creating a `Reading` model instead of specific models for each sensor type. Then for each property type of the sensor, a `Reading` is created, e.g., three `Reading`s for accelerometer data and one `Reading` for heart rate data. Listing 6.25 shows the model class for `Reading` which has properties for the session it belongs to, the sensor type, a timestamp, reading type, and the reading value itself. It should be noted that models in Realm do not require primary keys. Therefore, our `Reading` model does not need a primary key.

Listing 6.25: Reading model in Nyx.

```
1  public class Reading extends RealmObject {
2
3      private long sessionId;
4
5      private long sensorId;
6
7      private long timeStamp;
8
9      private String readingType;
10
11     private float data;
12
13     // Getters and setters omitted for brevity
14 }
```

A `Sensor` model is created to store the name of the sensors along with a primary key. Additional properties may be added, but for the purposes of our goal, we do not see it as necessary. Listing 6.26 exemplifies the `Sensor` model.

Listing 6.26: Sensor model in Nyx.

```
1  public class Sensor extends RealmObject {
2
3      @PrimaryKey
4      private long sensorId;
5
6      private String sensorName;
7
8      // Getters and setters omitted for brevity
9  }
```

The `Session` model stores attributes related to a data acquisition session and is shown in Listing 6.27. A `Session` has both a primary key and a UUID string. The reason for this is that Hypnos does have any knowledge of the database within Nyx and therefore needs a randomly generated UUID to add readings to a session. Also, if we were to store a UUID value for each `Reading` would use more storage than if an integer is used. Therefore, we use two identifiers for a single `Session`. This way, Hypnos does not need to know anything regarding the database in Nyx. At the same time, we save storage space for longer running sessions that generate millions of `Reading`s.

Also stored in a `Session` is the start and end times of a session, `startTime` and `endTime`, the number of readings reported from Hypnos, `readingsCount`, and a device identifier for the model of the Fitbit smartwatch, `deviceModel`. Since disconnections may occur during a session, the number of `Reading` stored in Nyx may be less than what is recorded in Hypnos. To determine whether data has been lost, we may compare the `readingsCount` property with the number of `Reading`s in the database for a given session.

Listing 6.27: Session model in Nyx.

```
1  public class Session extends RealmObject {
2
3      @PrimaryKey
4      private long sessionId;
5
6      @Required
7      private String uuid;
8
9      private long startTime;
10
11     private long endTime;
12
13     private int readingsCount;
14
15     private String deviceModel;
16
17     // Getters and setters omitted for brevity
18 }
```

Listing 6.28 shows an additional model, `SessionSensor`, which is used for storing the sensor configuration of a `Session`. Using this model, we are able to list the sensors that were enabled and the frequencies at which the sensors operated during a session. Therefore, it contains the primary keys of the `Session` and the `Sensor` as well as a property for the frequency.

Listing 6.28: SessionSensor model in Nyx.

```
1  public class SessionSensor extends RealmObject {
```

```
2
3       @PrimaryKey
4       private long id;
5
6       private long sessionId;
7
8       private long sensorId;
9
10      private float frequency;
11
12      // Getters and setters omitted for brevity
13  }
```

RealmNyxSessionStore

`RealmNyxSessionStore` is an implementation of the `NyxSessionStore` interface. The implementation overrides the methods and performs appropriate transactions to the Realm database. When Nyx receives the `INIT_SESSION` command, a new `Session` object is created using the model defined in Listing 6.27. At this time, no information regarding the start time, end time, or the number of readings is stored as the session is not yet performed. The `INIT_SESSION` command contains a list of enabled sensors and the frequencies. A check is performed to identify new sensors that are not already present in the database. If the sensors of a session already exist, no sensors are added to the database. Instead, `SessionSensors` for each `Sensor` are created using the frequencies.

Listing 6.29: Adding new readings to the Realm database

```
1   @Override
2   public void addReading(String sessionIdentifier, String sensorIdentifier,
3       List<Reading> readings) {
4
5       Realm realm = Realm.getDefaultInstance();
6
7       long sessionId = realm.where(Session.class).equalTo("uuid", sessionIdentifier)
8       .findFirst()
9       .getSessionId();
10
11      long sensorId = realm.where(Sensor.class).equalTo("sensorName",sensorIdentifier)
12      .findFirst()
13      .getSensorId();
14
15      for (Reading r : readings) {
16          r.setSensorId(sensorId);
17          r.setSessionId(sessionId);
18          queue.add(r);
19      }
20
```

```
21      if (queue.size() > BATCH_INSERT_THRESHOLD) {
22          List<Reading> toInsert = new ArrayList<>();
23
24          for (int i = 0; i < BATCH_INSERT_THRESHOLD; i++) {
25              toInsert.add(queue.poll());
26          }
27
28          try {
29              realm.executeTransactionAsync(r ->
30                  r.insert(toInsert)
31              );
32          } catch (Exception e) {
33              e.printStackTrace();
34          } finally {
35              realm.close();
36          }
37      }
38      realm.close();
39  }
```

Inserting new data to the Realm database is done by overriding the `addReading` method which takes the session UUID, sensor name, and a list of readings as parameters. The implementation of this method for `NyxSessionStore` is shown in Listing 6.29. For each invocation of this method, the corresponding `Session` and `Sensor` objects are fetched and their primary keys are added to each `Reading`. The number of readings at each invocation depends on the sensor frequency, sensor batch size, and the rate of dispatch to Nyx. Hypnos is configured to dispatched data each minute, meaning that a higher sensor frequency and sensor count will lead to this method being called recurrently. Each call to the `addReading` method opens an instance of the database, inserts data, and closes when the insertion is finished. Since opening and closing the database is a costly operation, we implement a queue mechanism to delay the insertion until a given number of readings are received. The `LinkedList` class of Java implements the `List<E>` interface, meaning that it has methods for adding and removing elements, but it also implements `Queue<E>`. Therefore, we may use a `LinkedList` as a FIFO queue and use the `poll` and `add` methods for queueing at one end and dequeuing at the other. Listing 6.30 shows a queue of `Reading` elements and `BATCH_INSERT_THRESHOLD` which determines the maximum size of the queue before data is persisted to the Realm database.

Listing 6.30: Instantiation of insertion queue

```
1  private LinkedList<Reading> queue;
2  private final int BATCH_INSERT_THRESHOLD = 5000;
```

6.3.5 Visualization

The acquired sensor data may be previewed in Nyx as graphs. After investigating the available chart libraries for Android, we decided to use an open-source [56] implementation. This library allows creating charts in various formats, e.g. line, bar, or pie chart. However, in our case, it is sufficient to show the data as a line chart with the timestamps on the x-axis and sensor readings on the y-axis.

Part IV

Evaluation and Conclusion

Chapter 7

Evaluation

7.1 Introduction

In this chapter, we perform several experiments to determine whether the applications developed in this theses are able to satisfy the requirements described in Chapter 5. The experiments should also guide the evaluation of the predefined aims of our thesis and to which degree they are attained.

7.2 Experiments

In this section, we perform experiments in which Hypnos and Nyx evaluated with respect to the requirements. We conduct four experiments that assess different aspects of our applications. In Experiment A, we evaluate the extensibility of Hypnos by adding a new sensor. Experiment B assesses the robustness with respect to longer data acquisition sessions by performing overnight sessions that are meant to approximate the typical use of our solution. Then in Experiment C, we test whether the data collected in Experiment B can be exported from the database. Finally, we evaluate in Experiment D the fault tolerance of Hypnos and Nyx when it comes to signal loss.

7.2.1 Experiment A: Extensibility

In this experiment, we investigate whether Hypnos may be extended with additional sensors. An understanding of the implementation of Hypnos is not a prerequisite as the sensors are defined independently in separate files. If the new sensor is created virtually, an additional step is needed to define the sensor using the `GenericSensor` class as shown in Section 6.1.6. Excluding this step, the procedure is identical for both physical and virtual sensors. The following is a step-by-step extension of Hypnos with a barometer sensor:

1. First, the sensor must be defined in the `SENSOR_DEFINITIONS` constant within `sensorDefinitions.js`. A sensor must be defined with properties seen in Listing 7.1, where `sensor` defines the identifier, `displayName`

defines the name displayed in Hypnos settings, `typeCount` defines the number of properties excluding the `timestamp`, and `types` defines the reading property names used in Nyx.

Listing 7.1: Barometer sensor in sensorDefinitions.js

```
{
    "sensor": "BAROMETER",
    "displayName": "Barometer",
    "typeCount": 1,
    "types": ["PASCAL"]
}
```

2. If the sensor is virtual, it must first be defined in a separate file inside the `app/sensor/virtual` directory similar to the existing sensors. If the sensor already exists, a new file should be created within the `app/sensor/sensors` directory with an appropriate name. This file should import and then instantiate the sensor within a function expression as shown in Listing 7.2. It is important that the `identifier` property is identical to the `sensor` property in the `sensorDefinitions.js` file since this property is used to bind the sensor configuration to the sensor instance. The last key, `properties`, must follow the same order as the `types` in `sensorDefinitions.js`, but should reference the actual sensor properties. For physical sensors, these properties are available in the SDK documentation, otherwise, it should be `"val"`. Since the barometer sensor has a property `pressure`, we use this property for `properties`.

Listing 7.2: Instantiation of the barometer sensor

```
import { Barometer } from "barometer";

export const barometer = () => {
    return {
        sensor: new Barometer(),
        identifier: "BAROMETER",
        properties: ["pressure"]
    }
}
```

3. The last step is to import the sensor file created in the previous step, in the file `sensorObjects.js` such that it may be utilized by the `SensorManager`. Listing 7.3 shows the `sensorObjects.js` with the barometer sensor. Installing and running the application should now display the barometer sensor in the application preferences of Hypnos. The barometer sensor may be used in a session without any further modifications to either Hypnos or

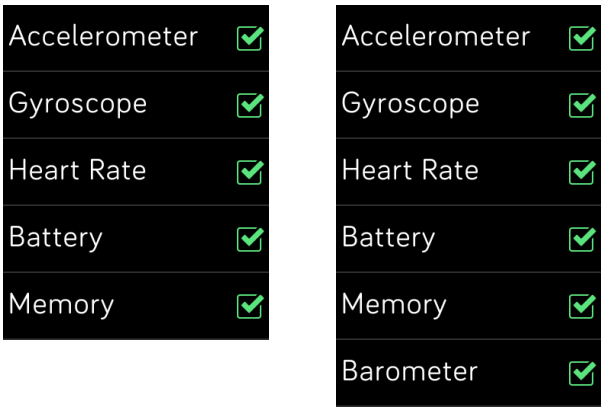
Nyx. To demonstrate that the sensor is working as intended and that the sensor readings are persisted in Nyx, we record a short session.

Listing 7.3: Barometer sensor within sensorObjects.js

```
1 import { accelerometer } from './sensors/accelerometer';
2 import { gyroscope } from './sensors/gyroscope';
3 ...
4 import { barometer } from './sensors/barometer';
5
6 export const SENSORS = [accelerometer, gyroscope, ... , barometer];
```

Results

After performing the steps listed in Experiment A, we should be able to toggle the sensor within Hypnos preferences as well as change its frequency. In Figure 7.1 (a) and (b), we see the preferences before and after the barometer sensor is added, respectively.

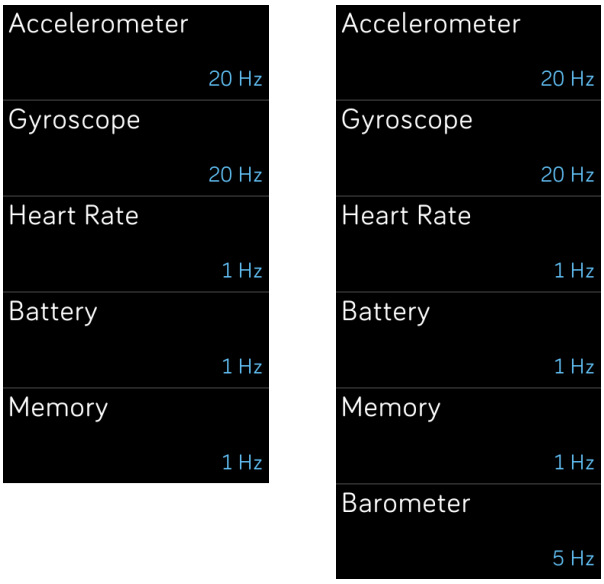


(a) Before experiment A (b) After experiment A

Figure 7.1: Sensor state settings in Hypnos

If the sensor remains enabled, it should collect data and dispatch it to Nyx when a new session is started. The frequency of the sensor may also be configured in application preferences. Figure 7.2 (a) and (b) shows the frequency settings before and after Experiment A, respectively. By default, a new sensor will always be configured to use a frequency of 1 Hz. The barometer sensor frequency may be changed by tapping the sensor and selecting the new frequency. In Figure 7.3, we can see that all of the enabled sensors which were active during the short session are stored in Nyx. Along with each sensor, the frequency which was used

is also listed. Tapping on any of the sensors in this list will show the recorded data. Since we added the barometer sensor, we check if the data from the sensor has been recorded. Figure 7.4 shows the sensor data from the barometer sensor as a graph.



(a) Before experiment A (b) After experiment A

Figure 7.2: Sensor frequency setting in Hypnos

Discussion

Experiment A demonstrates that Hypnos is extensible with regard to integrating novel sensors for data collection. The process requires only three steps (excluding the step for virtual sensors) that are independent of the implementation. The new sensor may be enabled or disabled or configured to use a different frequency after it has been added, without any additional work. Further, no modifications are needed in Nyx to either store the data from the new sensor. Therefore, this experiment verifies that our implementation ensures that minimal effort is required to extend Hypnos with novel sensors. Following the same procedure, we may add the oximeter sensor when it is enabled.

Although the procedure is effortless, we cannot guarantee that the memory of the Ionic smartwatch will not be a limitation when recording data with multiple sensors at higher frequencies. Since data from each sensor is temporarily

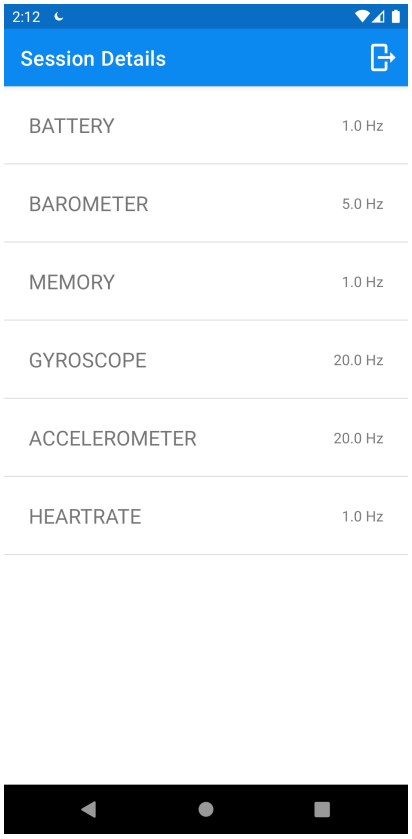


Figure 7.3: Session details of Experiment A

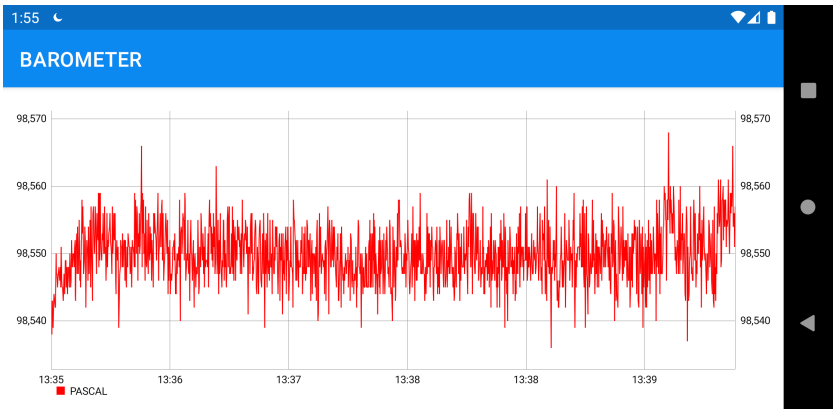


Figure 7.4: Barometer sensor data in Nyx

managed in memory, e.g., updating time stamps and writing to disk, the 64 kilobytes of memory may not be enough for more than six sensors. However, newer smartwatches from Fitbit that also have built-in oximeters, e.g., Versa 2, have 128 kilobytes of memory [44], which diminishes this concern.

7.2.2 Experiment B: Longer Session

Since Hypnos should be able to record longer, overnight sessions, we conduct an experiment where the duration is above our requirement of 8 hours. During a long session with multiple sensors collecting data, the battery may deflate, signals may be lost or the Android application may not persist the sensor data. To verify that Hypos may be used for collecting sensor data during overnight sessions, we perform an experiment using multiple sensors at various frequencies. Since the connected Android device will be running Nyx simultaneously, we must also ensure that Nyx is able to persist the data without affecting the battery life significantly.

Android Test Device

The specifications of the Android device used in this experiment are listed in Table 7.1. This device was launched around the same period as the Ionic smartwatch. Since our test device falls under the tablet category, it has a larger battery capacity than what is typical for a smartphone. However, since we are mainly interested in the battery usage of Hypnos, the battery capacity of the Android device is insignificant. Although the battery is larger, the other characteristics, e.g., memory, Bluetooth version, and Android version, are comparable to an ordinary smartphone. We therefore consider our testing device as representative of a typical Android smartphone.

Experiment Setup

This experiment should assess whether we are able to continuously record a sleep session lasting approximately 8 hours. In a typical monitoring session, the oximeter sensor should be used for collecting oxygen saturation data, an accelerometer sensor for tracking movement, and a heart rate sensor. To perform the longer experiment, we should therefore enable at least three sensors to verify that our applications may be used for the case mentioned above. We perform this experiment with five sensors, specifically, the accelerometer, gyroscope, barometer, heart rate, and battery sensor. Except for the battery, which is a virtual sensor used for logging the battery usage, all other sensors are built into the smartwatch.

Table 7.1: Specifications of the Android test device

Device	**Samsung Galaxy Tab A**
Model	Samsung SM-T580
Released	2016
SoC	Samsung Exynos 7870
CPU	1.6 GHz Cortex-A53
Battery (mAh)	7300
Memory (GB)	2
Storage (GB)	16
Bluetooth	4.1
Android version	8.1.0

Therefore, the number of sensors used in this experiment should demonstrate that Hypnos may be used for acquiring sensor data from the oximeter sensor during sleep for detecting OSA.

Configuring the sensors at different frequencies may affect the resource usage of Hypnos, the throughput of the data transfer, and database performance in Nyx. Thus, we should vary the frequencies and evaluate whether the solution is able to handle such changes. The configurations of the frequencies are listed in Table 7.2. For each of the configurations, we perform three runs in order to confirm that the results are valid. Prior to each run, we ensure that no other process is running in the background on both devices, that other radio antennas (e.g., WiFi and GPS) are disabled, and that there are no surrounding Bluetooth-enabled devices that may interfere with the devices. The starting battery percentages of both the smartwatch and the Android device prior to the experiment are varied to reflect the typical usage of Hypnos for data acquisition. Therefore, we do not fully charge the smartwatch before each run.

Results

The results for both configurations are listed in Table 7.3. As can be seen, the durations for Configurations 1 and 2 were on average 29 959.7 and 29 843.3 seconds, respectively, which correspond to roughly 8 hours and 16 minutes. Although the duration is similar, the number of readings differs significantly between the configurations, i.e., the number of readings collected during Configuration 2 is more

Table 7.2: Sensor configurations used in Experiment B

Sensor	**Configuration 1**	**Configuration 2**
Accelerometer	20 Hz	40 Hz
Gyroscope	20 Hz	40 Hz
Barometer	5 Hz	5 Hz
Heart Rate	1 Hz	1 Hz
Battery	0.1 Hz	0.1 Hz
Accumulative	**46.1 Hz**	**86.1 Hz**

Table 7.3: Results from Configuration 1 and 2

	Mean	**SD**	**Min**	**Max**
Configuration 1				
Duration (s)	29 959.7	183.7	29808	30218
Readings	1 473 885.7	8997.5	1 466 108	1 486 496
Ionic Battery Usage (%)	27.3	2.1	25	30
Android Battery Usage (%)	3.7	1.7	2	6
Configuration 2				
Duration (s)	29 843.3	3.4	29839	29846
Readings	3 212 073.3	3548.3	3 207 802	3 216 490
Ionic Battery Usage (%)	39.7	3.8	37	45
Android Battery Usage (%)	5.7	0.9	5	7

than twice the number of readings collected during Configuration 1. Furthermore, the battery usage of the Ionic smartwatch was, on average, 27.3% for Configuration 1 and 39.7% for Configuration 2. In other words, there was an increase of approximately 50% in battery usage between the two configurations. A similar pattern was registered for the battery usage of the Android device. On average, the battery usage for Configuration 1 was 3.7%. For Configuration 2, the battery usage was on average 5.7%, i.e., an increase of approximately 39%. Surprisingly, the battery usage for the Android device running Nyx did not increase as much as for the Ionic smartwatch, despite the frequencies being almost doubled in Configuration 2.

The collected sensor data from all iterations of both configurations were available in Nyx. In Figure 7.5, we show the data collected from the heart rate sensor during one of the sessions in Experiment B. As can be seen, Hypnos was able to collect heart rate data without any apparent disruptions in the data. Similarly, Figure 7.6 shows the data collected from the accelerometer sensor.

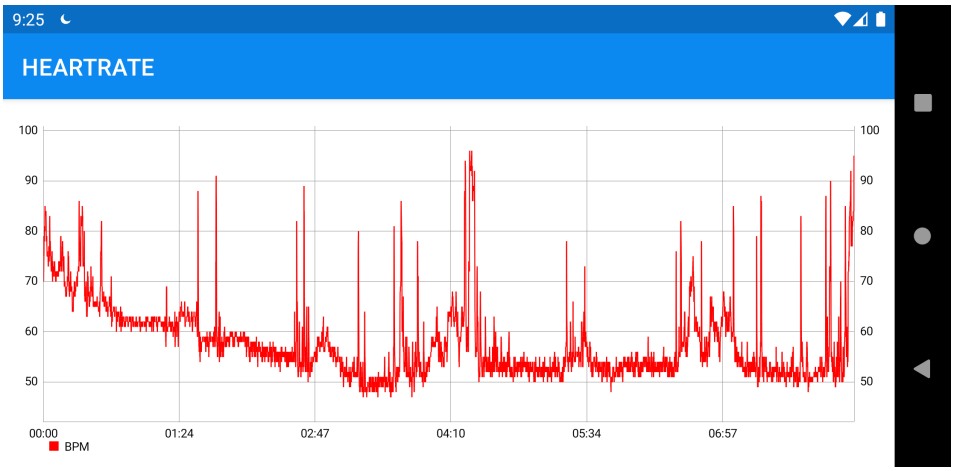


Figure 7.5: Heart rate sensor data from Experiment B

Table 7.4: Expected number of readings for Configuration 1 and 2

Accumulative Frequency	Expected Readings	Actual Readings
46.1 Hz	1 381 156	1 473 886
86.1 Hz	2 569 396	3 214 209

Discussion

Since the longer sessions were ended roughly at the same time, we may compare the results with regard to the collected readings and battery usage. The most noticeable difference between the configurations is that the number of readings increased approximately 120% from Configuration 1 to 2. An increase is to be expected since the frequency of both the accelerometer and gyroscope were increased from 20 to 40 Hz. However, the increase in the number of readings is significantly more than the increase in frequencies. The accumulative frequency of all sensors increased from 46.1 Hz to 86.1Hz, or close to 87%. We expect the increase in frequency to be a direct cause of the increase in readings registered. However, it appears that the number of readings increased more than the frequency. To illustrate this further, we estimate the number of readings expected by multiplying the accumulative frequency of a configuration with the average duration in Table 7.4. To our surprise, the number of acquired readings was significantly larger than our estimates. The difference is more significant when the accumulative frequency is higher, which suggests that the inconsistency is mainly caused by the accelerometer, gyroscope, or both.

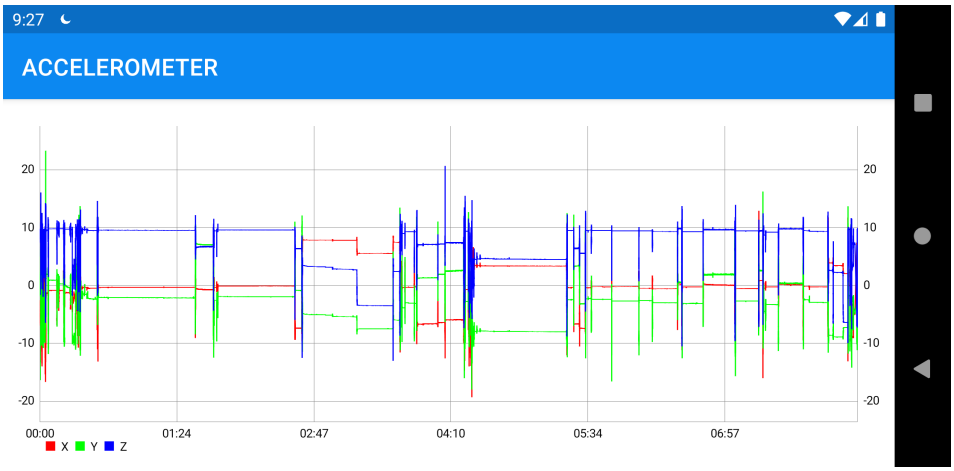


Figure 7.6: Accelerometer sensor data from Experiment B

Our initial assumption was that an incorrect behavior in either the handling, dispatch, or the persistence of the sensor data resulted in the same values being registered more than once, thereby increasing the total amount of readings. However, after inspecting the source code of both Hypnos and Nyx, we were not able to identify any mistake or error that may have led to this increase. A possible explanation is that the sensors are not operating at *exactly* the frequencies they are configured at prior to a new session. In Configuration 2, the accelerometer and gyroscope were both set to 40 Hz, but it is likely that the operational frequency was greater than 40 Hz. To put this theory to test, we exported the sensor data from Nyx and inspected the timestamps of the readings. Table 7.5, lists an excerpt of the accelerometer data. As can be seen, the differences between the timestamps are 20 milliseconds, which suggests that the actual frequency was closer to 50 Hz. After inspecting the SDK documentation further, we discovered that the barometer sensor operates at specific frequencies which are the closest to the configured values. Although the same behavior is not listed for the accelerometer and gyroscope sensors, we suspect that they are operating similarly to the barometer sensor. Similar results are encountered in [11] where the Flow sensor, an affordable respiratory belt, is benchmarked against the NOX T3. The Flow sensor, although being configured to sample at 10 Hz, is operating at varying frequencies below and above the configured value. Therefore, we presume that the behavior of the accelerometer sensor in our experiment is common among affordable sensors.

From the results, we see that the battery usage increases proportionally to the accumulative frequency of the sensors. An increase in frequency means that

Table 7.5: Sample from the accelerometer sensor data from Configuration 2

sessionId	sensorId	timeStamp	readingType	data
1	4	1620650411158	X	-0.2633695602416992
1	4	1620650411178	X	-0.37230968475341797
1	4	1620650411198	X	-0.32202911376953125
1	4	1620650411218	X	-0.20111846923828125
1	4	1620650411238	X	-0.2549896240234375

each batch will yield an increased amount of data to be processed, which in turn increases memory usage of the smartwatch and the data that should be transferred to the Android device. Therefore, the increase in battery usage is not surprising. However, despite the fact that the duration of Configuration 2 yielded a standard deviation of 3.4 seconds, the standard deviation for battery usage is 3.8%. In other words, the duration is close to identical but the reported battery usage suggests a deviation of approximately 4 percent. A possible explanation for this might be that the starting percentage of the battery was not identical in the multiple runs. In fact, after inspecting the results closer, we see a clear pattern: the higher battery percentage at the start of a session, the less the battery usage is reported after a session. The exact battery percentages may be seen in Appendix B.1. It appears that even though the battery usage may be identical, the reported battery percentages acquired using the SDK are non-linear. Since the SDK does not allow access to the exact usage in mAh, we are not able to determine whether the resource usage is identical or not. In the literature, we were not able to find any research assessing battery usage as percentages.

As the results of Experiment B demonstrate, Hypnos is able to record data from multiple sensors for longer than 8 hours. The configurations used in this thesis suggests that Hypnos may be used for recording data at frequencies up to 86.1 Hz. Since there were no disconnections during any of the recordings, we assume that the battery is the limiting factor when it comes to the duration of a session. Similarly, Nyx is able to persist data collected using the frequencies used in this experiment. However, these results do not prove that our applications may be used for collecting data at higher frequencies or with additional sensors. Even though we cache the sensor data on disk, the memory of the Ionic is a key limitation when a larger amount of readings are managed in memory.

7.2.3 Experiment C: Sensor Data Export

Although the collected sensor data stored in Nyx may be extracted and analyzed, the scope of our thesis does not involve analyzing and benchmarking the accuracy of the sensors. Instead, we perform an experiment to benchmark the duration of

Table 7.6: Export duration and file size for both configurations of Experiment B

	Mean	**SD**	**Min**	**Max**
Configuration 1				
Export Time (s)	165.9	2.4	161.9	167.8
File Size (MB)	29.7	1.0	28.4	30.6
Configuration 2				
Export Time (s)	407.2	13.7	388.7	421.4
File Size (MB)	65.9	2.7	62.2	68.7

exporting the data acquired in Experiment B. In the database within Nyx, readings from all sensors are stored in the same entity such that readings from novel sensors may be added to the same entity without modification of the database schema. Our implementation first separates the readings from each sensor into individuals collections. This procedure requires that data is loaded into memory transformed into a JSON file which may be transferred to a computer, e.g., using Bluetooth or email. In our experiment, we do not include the duration of the transfer of the JSON file from Nyx to a computer since the method of choice greatly affects the time.

In order to time the data exports correctly, we use the `currentTimeMillis` method of the `System` library of Java. This method registers the milliseconds elapsed since January 1, 1970. By registering the time prior to and after export, we are able to calculate the difference in milliseconds, which corresponds to the time it takes to query the data and build a JSON file. This approach ensures that there is no extra delay added, e.g., if the time was measured externally using a stopwatch. Since each configuration consists of three runs, we time the export of each. Additionally, we register the size of the exported data to evaluate the storage needs for the different configurations.

Results

The results from Experiment C are presented in Table 7.6, which contains the export time in seconds and the size of the exported file in megabytes. The export time for the first configuration is 165.9 seconds on average, whereas the export time for the second configuration is more than double the export time at 407.2 seconds. A similar pattern is observed for the size of the exported file. The exported file was on average 29.7 megabytes for Configuration 1 and 65.6 megabytes for the second configuration.

Discussion

The increase in the file size from Configuration 1 to 2 is approximately 122%, which is similar to the increase of readings at roughly 120%. Furthermore the average export time for the second configuration is approximately 145% longer than the first, despite the increase in the number of readings being around 120%. An increase in export time is expected as it takes more time to construct the JSON file due to the greater number of readings. However, the substantial increase in export time cannot be solely explained by the number of readings. It is likely that another parameter is affecting the export of data. Our implementation loads the data into memory and then constructs a JSON file. We suspects that, when the sensor data is loaded into memory simultaneously as the JSON file is constructed, our test device is running low on memory. In that case, the device must clear unused memory to allocate for the data export, which ultimately increases the time of export. The standard deviation of Configuration 1 is 2.4, whereas the same parameter is 13.7 for Configuration 1. This indicates that the export times vary significantly when the number of readings are large, which may in turn depend on the available memory on the device at the time of export.

The results of this experiment demonstrate that we are able to export the recorded sensor data from longer sessions lasting over 8 hours, regardless of the configuration used during data collection. However, the results do not prove that we are able to export data of sessions that are longer than 8 hours, use a higher accumulative frequency, or both. Nevertheless, for the purposes of this thesis, which is to export acquired data using similar configurations as in experiment B, we conclude that Nyx is able to extract the data in a reasonable time.

7.2.4 Experiment D: Fault Tolerance

A typical use for Hypnos is to acquire data throughout an overnight session from patients that may suffer from OSA. During such sessions, it is likely that the Bluetooth signal between the devices may be partially or completely obstructed, results in a disconnection. Disconnections may occur if the patient is wearing the smartwatch is visiting the toilet and the Android device is left behind or if the signals are obstructed by the body. In Experiment B, we assessed the quality of the applications with regard to longer recording sessions. During that experiment, we did not register any disconnects between the devices due to the devices being roughly 1 meter apart. However, our solution should be able to reconnect if the connection is lost and buffer data locally during the disconnected period.

In this experiment, we assess the fault tolerance of Hypnos and Nyx with respect to connection loss. Since the devices do not disconnect when they are in proximity, we simulate a disconnection by purposefully bringing them apart until the connection is lost. Furthermore, we should vary the duration between separating and unite the devices to figure out whether the disconnection time affects the reconnection. We test the reconnection after 1 and 5 minutes of connection loss to simulate shorter disruptions, i.e., visits to the toilet, but also 10 and 30 minutes to simulate longer disruptions, i.e., the patient is laying on the smartwatch during asleep.

Experiment Setup

Prior to the experiment, we attempted to find a location at which the connection was fully obstructed. For this, we used an Android application [57] which shows the signal strength between the Android device and a connected Bluetooth device. Furthermore, the sensors were configured identically to Configuration 1 for all durations. To ensure the validity of the results, we performed each duration three times. For each iteration, the session was started when the devices were in proximity and moved apart after approximately 2 minutes after a session is started. We started measuring the time after the connection had been lost and brought the devices together after the time had passed. For the transfer of the buffered data to complete, we made sure to wait roughly 2 minutes after the devices were united.

Results

The results from Experiment D are summarized in Table 7.7. We observed that the Bluetooth connection was re-established regardless of the disconnection duration. However, the time it takes for the devices to reconnect after unification varies depending on the disconnection duration. To calculate the reconnection time, we subtract the test duration from the total disconnection time, e.g., if the total disconnected time is 31 minutes and the test period is 30 minutes, the reconnection time is 60 seconds. Disconnections lasting 1 minute reconnected after 60.7 seconds on average, but as the standard deviation also indicates, the reconnection times varied between the iterations. Furthermore, the disconnections lasting 5 minutes reconnected the fastest at and most consistently at 11.4 seconds on average. For the longer disconnections lasting 10 and 30 minutes, the reconnection times were 32.0 and 23.4 seconds, respectively.

To determine whether data has been lost, we compared the number of sensor readings reported by Hypnos with the number of readings present in the database.

Table 7.7: Results from Experiment D

Duration (min)	**Reconnection Time (s)**		**Data Loss**		
	Mean	**SD**			
1	60.7	43.2	No	No	No
5	11.4	5.5	No	No	Yes
10	32.0	20.0	Yes	Yes	Yes
30	23.4	18.0	Yes	Yes	Yes

Looking at the results, we observe a correlation between the duration of the disconnection period and the loss of data. The experiment lasting 1 minute did not experience any loss of data. On the other hand, the results were variable for the 5-minute experiment which resulted in data loss for two out of three iterations. Figure 7.7 shows data from the barometer sensor during one of the 5-minute tests. Although there is no data loss in this iteration, the period of disconnection is clearly visible due to the variable height recorded by the barometer. Since the movement of the smartwatch introduces a height difference, the disconnected time is visible in the data. Furthermore, the disconnections lasting 10 and 30 minutes had noticeable data loss during the disconnected period in all iterations. Figure 7.8 shows the recorded barometer data from one of the 30-minute experiments. The spike at the beginning of the graph indicates the time at which the devices were separated and the connection was lost. Unlike the 5-minute experiment, Hypnos was not able to buffer the entire disconnection period. Approximately the first 5 minutes were buffered but the rest of the data is lost. As can be seen, the connection occurs after the 30 minutes have passed and the data acquisition continues.

Discussion

The results from Experiment D demonstrate that Hypnos and Nyx are able to re-establish a connection regardless of the disconnection periods that were tested. Whether they are able to reconnect after disconnections lasting more than 30 minutes is not investigated. Since the smartwatch periodically synchronizes with the Android device to transfer physical activity data [58], it is likely that they will reconnect irrespective of the disconnection time. Our results suggest that we should not experience issues regarding the reconnection after shorter disconnections, e.g., patients leave the Android device for bathroom visits during the night, or longer disconnections, e.g., obstruction of signals by the body.

Considering the time it takes for the devices to reconnect, we notice that the shortest duration of 1 minute takes the longest. On the other hand, the

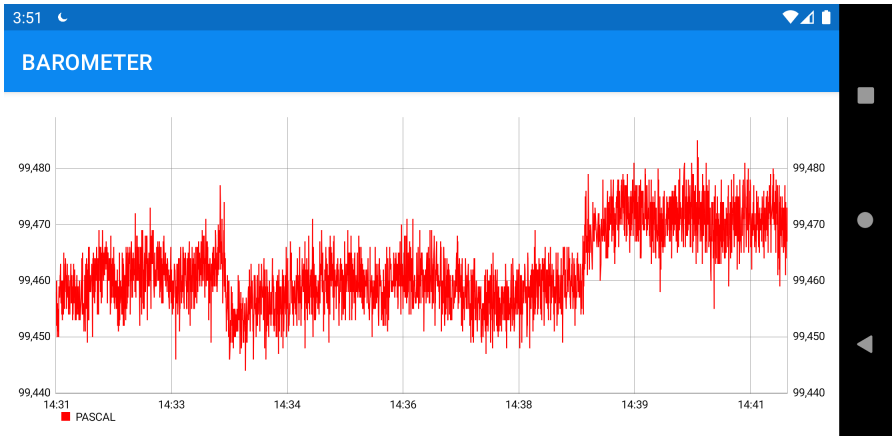


Figure 7.7: Barometer sensor data from 5 minute experiment

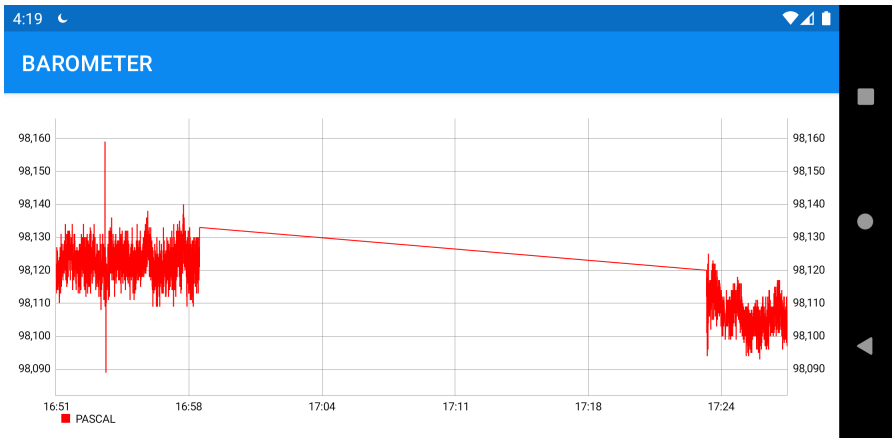


Figure 7.8: Barometer sensor data from 30-minute experiment

reconnection times for the longer disconnections is, in the worst case, 32 seconds. We expected shorter disconnections to reconnect faster but the opposite is true. Although the average reconnection time was 60.7, only one of the iterations deviated from the rest. As can be seen in Appendix B, two of the iterations took approximately 30 seconds and only one of the iterations took more than 2 minutes to reconnect. Therefore, it increases the average reconnection time.

Although the devices are able to reconnect in all cases, the loss of sensor data varies between the tested durations. For the 1-minute tests, there is no data loss in any of the iterations. The implementation of Hypnos ensures that files containing sensor data that are not sent to Nyx are stored on the disk until the next dispatch round. Therefore, despite the 1-minute disconnection, the buffered files are sent when the connection is available. On the other hand, we noticed significant data loss for the longer disconnections. One of the 5-minute tests also experienced data loss despite the relatively short duration. Data loss during the disconnections lasting 30 minutes is expected as Hypnos is not supposed to buffer more than a certain number of files. However, the results are suggesting that we are not able to buffer more than around 5 minutes.

Inspecting the issue further, we noticed that the maximum file count of the DispatchManager was set to 40 files, i.e., at most 40 files containing sensor data is buffered in Hypnos (see Section 6.1.7). To our surprise, increasing this number did not yield any difference in the buffering capacity of Hypnos. To identify the cause, we performed another experiment but his time logging the number of files in the dispatch queue. We noticed that the number of buffered files was maxing at 20. It appears that the File-transfer API is only able to enqueue the first 20 files and ignores files after reaching this limit. Since we use this File-transfer API to send the sensor files to the companion and verify whether the files are received, we are therefore only able to buffer the first 20 files on the smartwatch. Buffering a longer period requires that we either (1) increase the number of batches written to each file, or (2) implement a custom verification of files being sent by asking the companion. For the purposes of this experiment, we attempt to double the amount of data to each file. Figure 7.9 shows a new iteration of the 30-minute test but with twice the number of batches written to each file. As can be seen, increasing the write limit increases the readings being buffered.

7.3 Evaluation of Requirements

In the previous sections, we performed experiments to determine whether the applications developed in this thesis are able to meet the requirements defined in Section 5.1. In this section, we evaluate our applications with respect to these requirements.

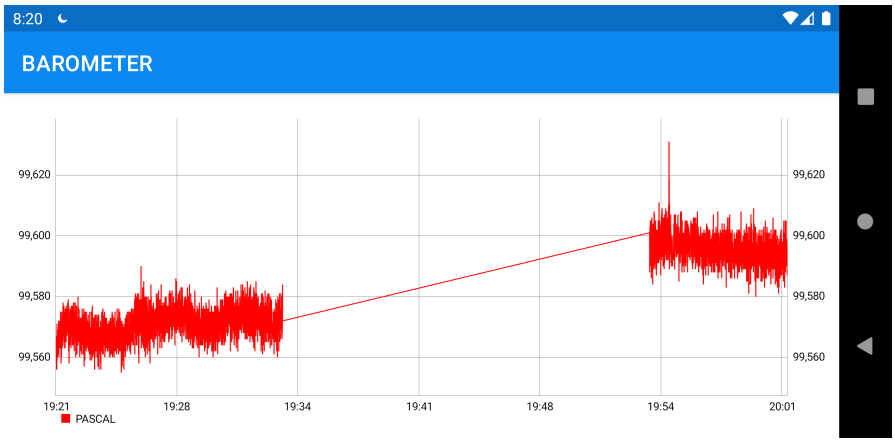


Figure 7.9: Updated barometer sensor data from 30 minute experiment

7.3.1 Evaluation of Non-functional Requirements

The non-functional requirements state that Hypnos should be extensible, resource-efficient, privacy-preserving, and simple. As demonstrated in Experiment A, Hypnos can be extended with new sensors. The process requires 3 steps in specific files that are not related to the implementation. Additional sensors can therefore be added without examining and understanding the source code. In addition, adding new sensors to Hypnos requires no changes to the database in Nyx. Therefore, we conclude that our applications are extensible.

Experiment B demonstrates that Hypnos is able to record sessions lasting over 8 hours and retain more than half of the battery life. Our results show that using a total frequency of 46.1 Hz retains more than 70% of the battery capacity. When the total frequency is 86 Hz, more than 60% of the battery capacity is left normal use of the Ionic smartwatch. In addition, since the sessions were able to collect data without any disruptions, we may assume that we do not exceed the memory limitations of the smartwatch. If we were, Hypnos would have stopped in the midst of the recording and the data collections would have not completed the 8 hours. Therefore, we conclude that Hypnos is resource-efficient since it is able to record overnight sessions despite the 64-kilobyte memory limitation and retain battery for typical smartwatch use.

In our solution, sensor data is transferred directly from Hypnos to the connected smartphone using Bluetooth. The data is then transferred to Nyx using a local WebSocket server and inserted into the database. This means that our applications function without requiring Internet connectivity. In fact, the sessions

in Experiment B are recorded with both WiFi and cellular disabled. The sensitive health data recorded with Hypnos never leaves the Android device and thereby ensures user privacy. Only if the user explicitly chooses to export the session data it may be transferred to an external location. Furthermore, performing a data acquisition session with Hypnos and Nyx requires only a couple of steps and no configuration. To connect the applications, it is enough to open Hypnos on the smartwatch and Nyx on the Android device, given that the FMA is running in the background. Configuration of the sensors can be performed directly within Hypnos. If not, a new session can be started. Therefore, we conclude that our applications fulfill the requirement regarding simplicity.

7.3.2 Evaluation of Functional Requirements

The functional requirements state that we should be able to add a new sensor, record longer data acquisition sessions, configure sensors, store and view the recorded sessions, and export the sessions to an external location. In the previous section, we have already considered the first three of these requirements. Therefore, we only address the last three functional requirements in this section.

In Experiment A, we added a barometer sensor to Hypnos and recorded a test session to verify the extensibility. We also demonstrated that the data may be viewed from Nyx after the session has been recorded. The recorded sessions in Nyx are organized as a list and selecting a session presents the sensors used in the session. Selecting a sensor will present the data visually as a graph with time labels at the bottom. Therefore, Experiment A demonstrates that we are able to store and preview the acquired sensor data in Nyx.

To export the sensor data, a dedicated export functionality is available in Nyx. Initiation an export creates a JSON file containing the data and brings up a menu for choosing the preferred method of sharing. In Experiment C, we evaluated the export functionality of Nyx by exporting the data collected in Experiment B. Since the sessions in Experiment B are all longer than 8 hours, this means that we are able to export sensor data from overnight recordings without any issues. We therefore conclude that our solution enables exporting data.

Chapter 8

Conclusion

8.1 Summary

In recent years, the availability and advancement of wearable technology has enabled the usage of such devices for continuous health monitoring. A variety of built-in sensors are able to measure various physiological metrics and evaluate health and physical activity. Of special interest is the pulse oximeter sensor which is able to measure the percentage of oxygenated blood. If the accuracy is adequate, a pulse oximeter may be used to detect breathing disruptions during sleep. The adoption of wearable technology with oximeter sensors may therefore lower the threshold for detection and treatment of OSA. Consequently, such sensors must be benchmarked against gold standards to evaluate the quality of the acquired data. The CESAR project aims to benchmark affordable sensors and smartwatches and investigates the accuracy of such devices.

In this thesis, we have developed a smartwatch application for the Ionic smartwatch, Hypnos, that is able to collect sensor data from the built-in sensors at various sampling rates. Sensors may be enabled or disabled depending on the needs of the data collection and the frequencies may be altered within the application preferences. The data is transferred to our Android application, Nyx, and stored persistently in a database. To ensure that the data transfer is reliable, we experimented with different techniques for caching and organizing the data on the smartwatch. Due to the highly limited memory of the Ionic smartwatch, we store data in binary files on the file system.

The Ionic smartwatch includes an oximeter sensor, but the use of this sensor is not available via the APIs at the time of our research. Instead, we use the available sensors to benchmark and evaluate our applications. In our evaluation, we perform experiments to ensure that the applications are able to meet the aims and goals of our thesis. Specifically, we perform an experiment to verify the extensibility with regard to additional sensors. The results of this experiment demonstrate that our solution can easily be extended with novel sensors. In addition, adding new sensors requires no knowledge of the implementation details of either Hypnos or Nyx as the changes are made in specific configuration files.

To ensure that the applications are able to collect data during sleep, we

perform an experiment collecting data from multiple sensors throughout longer sessions. Two distinct configurations with different frequencies were used to measure the battery usage of Hypnos. The results indicate that the applications may be used to collect data from multiple sensors with an accumulated frequency of up to 86.1 Hz for over 8 hours. Furthermore, an experiment is performed to evaluate whether the data collected from longer sessions may be exported to a desktop computer for further analysis. Our results demonstrate that Nyx is able to export the longer sessions recorded in this thesis as JSON files. Finally, we evaluate the fault tolerance of our applications with respect to connection loss. We observed that the applications were able to reconnect regardless of the various disconnection times that were tested.

8.2 Contributions

As a contribution to the CESAR project, we have investigated the development environment of the Fitbit smartwatches and the possibility of using the Fitbit Ionic smartwatch in health monitoring related to OSA. Our research and the applications developed in this thesis are meant to lay the foundation for the collection of sensor data from the Ionic smartwatch as well as newer smartwatches from Fitbit. Our applications may be improved, extended with additional sensors, or used as-is to evaluate the oximeter sensor and the possible use of this sensor for detecting breathing disruptions. Additional functionality and improvements for future versions are proposed in Section 8.4.

8.3 Open Problems

During the initial experiments of the data transfer, we observed that the companion application running within the FMA would exit arbitrarily, shortly after starting the sessions. Our brief investigation of this issue suggests that the Bluetooth module of our test device closes due to an unidentified error. As a result, the FMA loses connection with the smartwatch. Although the data is cached temporarily, the reconnection does not occur. Manually disabling and enabling the Bluetooth does not yield the same behavior, as the reconnection occurs almost instantaneously. Resetting the Bluetooth settings and clearing the cache of the Bluetooth module on the Android device appears to fix the issue. However, we were are not able to determine whether this issue is limited to the Bluetooth module of our device. A closer inspection must be performed to determine the cause of this issue.

8.4 Future Work

Although Hypnos and Nyx may be used to collect sensor data as-is, there are several improvements that have not been prioritized during our research. The following are suggestions for additional functionality that may be implemented in future versions of Hypnos and Nyx:

- **Real-time Data Analysis:** Currently, our implementation persists the sensor data in a database throughout a session. As a result, the data is accessible only after the session has been ended. An alternative mode for transmitting the data continuously, would enable us to process the data in real-time and detect certain events based on specific readings from the sensors. For example, the data from the oximeter sensor may be used to detect apneic events which may be used to evaluate the severity of the apneas. Furthermore, the accelerometer and heart rate sensor may be used to detect sleep cycles and evaluate restlessness and quality of sleep.
- **Additional Preferences in Hypnos:** The current version of Hypnos allows individual sensors to be enabled or disabled and configured with regard to the sampling rate. There are, however, other parameters that could be configured within the application preferences. One example is the batch rate of the sensor data, i.e., how often the events are delivered from the sensors and processed. In the current implementation, we use a step function that sets the batch size to 1 for frequencies below 1 and twice the frequency for frequencies above 1. Since this parameter directly affects memory usage, it should be easily available in the preferences. Another example is the dispatch rate of the sensor data from Hypnos to Nyx which, in our current implementation, is fixed at 1 minute. Such parameters should be easily configurable within Hypnos to allow our application to be configured for various needs.
- **SDK Upgrade:** Hypnos uses the latest available SDK for the Ionic smartwatch, version 4.2.1. The latest SDK, version 5.0, is only available for the newer Fitbit smartwatches. In order to run Hypnos on these devices, parts of the source code must be altered. However, the changes are not directly related to our implementation since the APIs used are still available, but the naming convention of UI files and certain UI components have changed. Since the new SDK is exclusive for newer Fitbit devices, a fork must be created to perform these changes such that Hypnos may be used to acquire sensor data from newer Fitbit smartwatches with novel sensors.

Bibliography

[1] Mayo Clinic Staff. *Obstructive sleep apnea*. June 2019. URL: https://www.mayoclinic.org/diseases-conditions/obstructive-sleep-apnea/symptoms-causes/syc-20352090 (visited on 05/26/2020).

[2] *Worldwide Wearables Market to Top 300 Million Units in 2019 and Nearly 500 Million Units in 2023, Says IDC*. Dec. 2019. URL: https://www.idc.com/getdoc.jsp?containerId=prUS45737919 (visited on 05/23/2020).

[3] *Heart health notifications on your Apple Watch*. Jan. 2021. URL: https://support.apple.com/en-us/HT208931 (visited on 01/25/2021).

[4] Morten Engstrøm et al. *Obstruktiv søvnapné*. Nov. 2015. URL: https://tidsskriftet.no/2015/11/klinisk-oversikt/obstruktiv-sovnapne (visited on 05/22/2020).

[5] Mayo Clinic Staff. *Polysomnography (sleep study)*. URL: https://www.mayoclinic.org/tests-procedures/polysomnography/about/pac-20394877 (visited on 05/13/2021).

[6] Juan F. Masa et al. "Effectiveness of home respiratory polygraphy for the diagnosis of sleep apnoea and hypopnoea syndrome". In: *Thorax* 66 (2011).

[7] Kaitlin A. Freeberg et al. "Assessing the ability of the Fitbit Charge 2 to accurately predict VO2max". In: *mHealth* 5 (2019). URL: https://mhealth.amegroups.com/article/view/29481.

[8] World Health Organization (WHO). *Global surveillance, prevention and control of CHRONIC RESPIRATORY DISEASES*. URL: https://www.who.int/gard/publications/GARD%20Book%202007.pdf (visited on 05/26/2020).

[9] Fredrik Løberg, Vera Goebel, and Thomas Plagemann. "Quantifying the Signal Quality of Low-cost Respiratory Effort Sensors for Sleep Apnea Monitoring". In: *Proceedings of the 3rd International Workshop on Multimedia for Personal Health and Health Care* (2018), pp. 3–11.

[10] Stein Kristiansen et al. "A Clinical Evaluation of a Low-Cost Strain Gauge Respiration Belt and Machine Learning to Detect Sleep Apnea". In: *CoRR* abs/2101.02595 (2021). URL: https://arxiv.org/abs/2101.02595.

[11] Stein Kristiansen et al. "Evaluating a Low-Cost Strain Gauge Breathing Sensor for Sleep Apnea Detection at Home". In: *4th International Workshop on IoT Enabling Technologies in Healthcare (IoT-Health 2021)* (2021).

[12] Helge Opdahl. *Oksygenmetning*. May 2018. URL: https://sml.snl.no/oksygenmetning (visited on 05/25/2020).

[13] Mayo Clinic Staff. *Hypoxemia*. Dec. 2018. URL: https://www.mayoclinic.org/symptoms/hypoxemia/basics/definition/sym-20050930 (visited on 05/17/2020).

[14] Amal Jubran. "Pulse oximetry". In: *Critical care* 19 (2015). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4504215/.

[15] Haakon Willem Hoel Bakker. "Sopor: An extensible watchOS application for sleep session recording". MA thesis. University of Oslo, 2020. URL: https://www.duo.uio.no/bitstream/handle/10852/78657/Sopor-An-extensible-watchOS-application-for-sleep-session-recording.pdf?sequence=1&isAllowed=y.

[16] Kenneth Aune Frisvold. "Non-Invasive Benchmarking of Pulse Oximeters - An Empirical Approach". MA thesis. University of Oslo, 2018.

[17] Felix Griffin Halvorsen. "Garmin smartwatches to detect desaturation events as part of OSA screening at home". MA thesis. University of Oslo, 2020.

[18] *Fitbit Launches Ionic, the Ultimate Health and Fitness Smartwatch*. URL: https://investor.fitbit.com/press/press-releases/press-release-details/2017/Fitbit-Launches-Ionic-the-Ultimate-Health-and-Fitness-Smartwatch/default.aspx (visited on 01/19/2021).

[19] URL: https://www.fitbit.com/us/products/smartwatches/ionic (visited on 05/21/2020).

[20] James Stables. *Fitbit Estimated Oxygen Variation: We explain SpO2, sleep apnea and blood oxygen*. Jan. 2020. URL: https://www.wareable.com/fitbit/fitbit-estimated-oxygen-variation-explained-7878 (visited on 01/20/2021).

[21] *Fitbit Finally Releases Blood Oxygen Monitoring*. Jan. 2020. URL: https://www.nasdaq.com/articles/fitbit-finally-releases-blood-oxygen-monitoring-2020-01-17 (visited on 01/20/2021).

[22] *ScanWatch*. URL: https://www.withings.com/de/en/scanwatch (visited on 01/19/2021).

[23] *ScanWatch - Frequently asked questions about electrocardiogram (ECG)*. 2020. URL: https://support.withings.com/hc/en-us/articles/360004559098-ScanWatch-Frequently-asked-questions-about-electrocardiogram-ECG- (visited on 01/21/2021).

[24] *Fitbit Ionic User Manual*. Fitbit. 2018. URL: https://images-na.ssl-images-amazon.com/images/I/C1lr3XRvF8S.pdf (visited on 05/16/2021).

[25] *Fitbit SDK Toolchain*. URL: https://github.com/Fitbit/fitbit-sdk-toolchain (visited on 01/25/2021).

[26] *Fitbit Web API Basics*. URL: https://dev.fitbit.com/build/reference/web-api/basics/ (visited on 03/08/2021).

[27] *About Node.js®*. URL: https://nodejs.org/en/about/ (visited on 05/24/2020).

[28] *Glossary of Terms*. URL: https://dev.fitbit.com/build/guides/glossary/#overview (visited on 05/26/2020).

[29] *File Transfer Guide*. 2019. URL: https://dev.fitbit.com/build/guides/communications/file-transfer/ (visited on 05/25/2020).

[30] *Application Architecture Guide*. URL: https://dev.fitbit.com/build/guides/application/ (visited on 02/27/2021).

[31] T. Editors of Encyclopaedia Britannica. "Android". In: (Aug. 2020). URL: https://www.britannica.com/technology/Android-operating-system (visited on 03/05/2021).

[32] *Members*. URL: https://www.openhandsetalliance.com/oha_members.html (visited on 03/08/2021).

[33] *Mobile Operating System Market Share Worldwide*. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 05/16/2021).

[34] *Understand Tasks and Back Stack*. URL: https://developer.android.com/guide/components/activities/tasks-and-back-stack (visited on 02/18/2021).

[35] *Services overview*. URL: https://developer.android.com/guide/components/services (visited on 03/05/2021).

[36] *Android 8.0 Behavior Changes*. URL: https://developer.android.com/about/versions/oreo/android-8.0-changes (visited on 05/16/2021).

[37] I. Fette and A. Melnikov. *The WebSocket Protocol*. URL: https://tools.ietf.org/html/rfc6455#page-4 (visited on 03/06/2021).

[38] *Build better apps, faster*. URL: https://realm.io (visited on 04/26/2021).

[39] *Features Of SQLite*. URL: https://www.sqlite.org/features.html (visited on 03/09/2021).

[40] *What is NoSQL?* 2021. URL: https://aws.amazon.com/nosql/ (visited on 04/10/2021).

[41] Mariusz Chmielewski, Damian Frąszczak, and Dawid Bugajewski. "Architectural concepts for managing biomedical sensor data utilised for medical diagnosis and patient remote care". In: *MATEC Web of Conferences* 210 (2018).

[42] Karim Kussainov and Bolatzhan Kumalakov. "Mobile Data Store Platforms: Test Case based Performance Evaluation". In: *Proceedings of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2016)* 3 (2016).

[43] *All About Fitbit Ionic*. Aug. 2017. URL: https://community.fitbit.com/t5/Ionic/All-About-Fitbit-Ionic/td-p/2121495 (visited on 03/11/2021).

[44] *Announcing Fitbit OS SDK 4.2*. URL: https://dev.fitbit.com/blog/2020-09-10-announcing-fitbit-os-sdk-4.2/ (visited on 05/13/2021).

[45] *Heart Rate Sensor Guide*. URL: https://dev.fitbit.com/build/guides/sensors/heart-rate/ (visited on 03/11/2021).

[46] Maarten Shellevis. "Getting access to your own Fitbit data". Bachelor's thesis. Radboud University, Aug. 2016. URL: https://www.cs.ru.nl/bachelors-theses/2016/Maarten_Schellevis___4142616___Getting_access_to_your_own_Fitbit_data.pdf.

[47] *Application Sandbox*. URL: https://source.android.com/security/app-sandbox (visited on 04/07/2021).

[48] Ed. T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Mar. 2014. URL: https://tools.ietf.org/html/rfc7159 (visited on 04/12/2021).

[49] *Messaging Guide*. URL: https://dev.fitbit.com/build/guides/communications/messaging/ (visited on 05/25/2020).

[50] *sdk-multi-view*. URL: https://github.com/Fitbit/sdk-multi-view (visited on 01/19/2021).

[51] *Ionic Views*. URL: https://github.com/gaperton/ionic-views (visited on 02/01/2021).

[52] *Companion Guide*. URL: https://dev.fitbit.com/build/guides/companion/ (visited on 04/15/2021).

[53] *Sensors API*. URL: https://dev.fitbit.com/build/reference/device-api/sensors/ (visited on 04/18/2021).

[54] *Optimize for Doze and App Standby*. Nov. 2020. URL: https://developer.android.com/training/monitoring-device-state/doze-standby (visited on 05/05/2021).

[55] *Java WebSockets*. URL: https://github.com/TooTallNate/Java-WebSocket (visited on 02/15/2021).

[56] Philipp Jahoda. *MPAndroidChart*. URL: https://github.com/PhilJay/MPAndroidChart (visited on 05/02/2021).

[57] Edgar García Leyva. *Bluetooth signal strength meter*. URL: https://play.google.com/store/apps/details?id=dev.egl.com.intensidadbluetooth&hl=en_US&gl=US (visited on 05/21/2021).

[58] *How do Fitbit devices sync their data?* URL: https://help.fitbit.com/articles/en_US/Help_article/1877.htm (visited on 05/23/2021).

Appendix A

Source Code

The source code for the applications developed in this thesis is openly available on Github as separate repositories. The applications may be used as-is or by forking the repositories and making the necessary changes.

A.1 Source Code of Hypnos

The source code of the smartwatch application is available at `https://github.com/ademsalih/hypnos`. Listing A.1 shows how to clone the repository of Hypnos.

Listing A.1: Cloning repository of Hypnos from Github

```
1 git clone https://github.com/ademsalih/hypnos.git
```

A.2 Source Code of Nyx

The source code of the Android application is available at `https://github.com/ademsalih/nyx`. Listing A.2 shows how to clone the repository of Nyx.

Listing A.2: Cloning repository of Nyx from Github

```
1 git clone https://github.com/ademsalih/nyx.git
```

Appendix B

Data from experiments

B.1 Experiment B

In this section, we list the data from Experiment B. The runs using Configuration 1 are listed in Table B.1 and the runs using Configuration 2 are listed in Table B.2.

Table B.1: Results from Experiment B using Configuration 1

Duration (s)	**Readings**	**Ionic Battery Start (%)**	**Ionic Battery End (%)**	**Android Battery Start (%)**	**Android Battery End (%)**
29808	1 466 108	88	61	59	57
29853	1 469 053	60	30	55	52
30218	1 486 496	100	75	70	64

Table B.2: Results from Experiment B using Configuration 2.

Duration (s)	**Readings**	**Ionic Battery Start (%)**	**Ionic Battery End (%)**	**Android Battery Start (%)**	**Android Battery End (%)**
29839	3 211 928	100	63	72	67
29846	3 216 490	71	26	66	61
29845	3 207 802	99	62	85	78

B.2 Experiment C

In this section, we list the data from Experiment C. The runs using Configuration 1 are listed in Table B.3 and the runs using Configuration 2 are listed in Table B.4.

Table B.3: Results from Experiment C using Configuration 1.

Duration (s)	**Readings**	**Export Time (s)**	**File Size (MB)**
29 808	1 466 108	161.9	28.4
29 853	1 469 053	164.9	30.2
30 218	1 486 496	167.8	30.6

Table B.4: Results from Experiment C using Configuration 2.

Duration (s)	**Readings**	**Export Time (s)**	**File Size (MB)**
29839	3 211 928	411.4	62.2
29846	3 216 490	421.4	68.7
29845	3 207 802	388.7	66.9

B.3 Experiment D

In this section, we present the data from Experiment D. The results from the 1 and 5-minute disconnections are listed in Table B.5 and Table B.6, respectively. The results from the 10 and 30-minute disconnections are listed in Table B.7 and Table B.8, respectively.

Table B.5: Results from Experiment D with 1-minute disconnections

Duration (s)	**Disconnection Duration (s)**	**Readings**	**Readings Registered**
557.9	181.7	27199	27199
418.4	90.1	20457	20457
333.6	90.2	16305	16305

Table B.6: Results from Experiment D with 5-minute disconnections

Duration (s)	**Disconnection Duration (s)**	**Readings**	**Readings Registered**
712.5	308.3	34748	34748
632.6	306.9	30698	30698
610.0	318.9	29592	29182

Table B.7: Results from Experiment D with 10-minute disconnections

Duration (s)	**Disconnection Duration (s)**	**Readings**	**Readings Registered**
1061.8	630.4	51853	35084
1013.9	657.4	49422	31013
985.1	608.3	47944	35398

Table B.8: Results from Experiment D with 30-minute disconnections

Duration (s)	**Disconnection Duration (s)**	**Readings**	**Readings Registered**
2199.1	1848.7	107921	34080
2173.0	1808.3	106746	33397
2175.7	1813.2	106888	33539